10/531287

PCT/JP03/12839

日本国特許庁
JAPAN PATENT OFFICE

07.10.03

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office.

出願年月日
Date of Application: ２００２年１０月１５日

出願番号
Application Number: 特願２００２－３００７３

[ST. 10/C]: [ＪＰ２００２－３００７３]

出願人
Applicant(s): 株式会社ルネサステクノロジ

２００３年１１月 ７日

特許庁長官
Commissioner,
Japan Patent Office 今井康夫

出証番号 出証特２００３－３０９２０５４

【書類名】　特許願

【整理番号】　H02014091

【提出日】　平成14年10月15日

【あて先】　特許庁長官殿

【国際特許分類】　G06F 19/00

【発明者】

【住所又は居所】　東京都小平市上水本町五丁目２０番１号　株式会社日立製作所　半導体グループ内

【氏名】　谷本　匡亮

【発明者】

【住所又は居所】　東京都小平市上水本町五丁目２０番１号　株式会社日立製作所　半導体グループ内

【氏名】　鎌田　丈良夫

【特許出願人】

【識別番号】　000005108

【氏名又は名称】　株式会社日立製作所

【代理人】

【識別番号】　100089071

【弁理士】

【氏名又は名称】　玉村　静世

【電話番号】　03-5217-3960

【手数料の表示】

【予納台帳番号】　011040

【納付金額】　21,000円

【提出物件の目録】

【物件名】　明細書　1

【物件名】　図面　1

【物件名】　要約書　1

【プルーフの要否】　要

【書類名】　明細書

【発明の名称】　コンパイラ及び論理回路の設計方法

【特許請求の範囲】

【請求項１】　所定のプログラム言語を流用して記述された第１プログラム記述を回路記述に変換可能なコンパイラであって、

前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含み、

前記回路記述は、前記第１プログラム記述が特定する回路動作を実現するハードウェアを所定のハードウェア記述言語で特定することを特徴とするコンパイラ。

【請求項２】　所定のプログラム言語を流用して記述された第１プログラム記述を所定のプログラム言語を用いた第２プログラム記述に変換可能なコンパイラであって、

前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含み、

前記第２プログラム記述は、前のサイクルの状態を参照可能にする為に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文とを含むことを特徴とするコンパイラ。

【請求項３】　所定のプログラム言語を流用して記述された第１プログラム記述を、所定のプログラム言語を用いた第２プログラム記述と回路記述に変換可能なコンパイラであって、

前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含み、

前記第２プログラム記述は、前のサイクルの状態を参照可能にする為に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文とを含み、

前記回路記述は、前記第２プログラム記述で定義されるハードウェアを所定の

ハードウェア記述言語で特定することを特徴とするコンパイラ。

【請求項４】　前記所定のプログラム言語はＣ言語であることを特徴とする請求項１乃至３の何れか１項記載のコンパイラ。

【請求項５】　前記ハードウェア記述言語はＲＴＬレベルの記述言語であることを特徴とする請求項１又は３記載のコンパイラ。

【請求項６】　タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する第１処理と、

前記第１プログラム記述に基づいて前記タイミング仕様を満足する回路情報を生成する第２処理と、を含むことを特徴とする論理回路の設計方法。

【請求項７】　前記第２処理は、前記第１プログラム記述を変換して、レジスタ代入文が入力変数と出力変数を用いて変形されると共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含む第２プログラム記述を前記回路情報として生成する処理を含むことを特徴とする請求項６記載の論理回路の設計方法。

【請求項８】　前記第２処理は、前記第２プログラム記述を変換して、前記タイミング仕様を満足するハードウェアを所定のハードウェア記述言語で特定するための回路記述を更に別の前記回路情報として生成する処理を含むことを特徴とする請求項７記載の論理回路の設計方法。

【請求項９】　前記プログラム言語はＣ言語であることを特徴とする請求項８記載の論理回路の設計方法。

【請求項１０】　前記第２プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含むことを特徴とする請求項９記載の論理回路の設計方法。

【請求項１１】　前記第２処理は、前記第１プログラム記述を変換して、前記レジスタ代入文が入力変数と出力変数を用いて変形された記述を含む第２プログラム記述を前記回路情報として生成する処理を含むことを特徴とする請求項６記載の論理回路の設計方法。

【請求項１２】　前記第２処理は、前記第２プログラム記述を変換して、前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含み、所定のプログラム言語で記述されてコンピュータで実行可能なむ第３プログラム記述を、前記回路情報として生成する処理を含むことを特徴とする請求項１１記載の論理回路の設計方法。

【請求項１３】　前記第３プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含むことを特徴とする請求項１２記載の論理回路の設計方法。

【請求項１４】　タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する入力処理と、

前記レジスタ代入文が入力変数と出力変数を用いて変形されると共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含み、前記所定のプログラム言語で記述された第２プログラム記述を生成する変換処理と、を含むことを特徴とする論理回路の設計方法。

【請求項１５】　前記変換処理は、第１プログラム記述に基づいてＣＦＧを生成する過程で、前記ＣＦＧに前記クロック境界記述に対応してクロック境界ノードを設定し、前記クロック境界ノードの後に、前記レジスタ代入記述を挿入することを特徴とする請求項１４記載の論理回路の設計方法。

【請求項１６】　第２プログラム記述に対してそのＣＦＧを利用しながらステート遷移毎の変数表を作成しながらコード最適化を行う最適化処理を更に含むことを特徴とする請求項１５記載の論理回路の設計方法。

【請求項１７】　前記変数表においてステート間で変数に変化のない部分を前置保持を要する部分として抽出し、抽出された部分に、出力変数に入力変数を代入する記述を追加する前置保持処理を更に含むことを特徴とする請求項１６記載の論理回路の設計方法。

【請求項１８】　前記前置保持処理を経た変数表の各ステート遷移毎の変数と引数に基づいてステートマシンを構成するコードの抽出を行う抽出処理を更に

含むことを特徴とする請求項１７記載の論理回路の設計方法。

【請求項１９】　前記抽出処理で抽出されたステートマシン構成コードと第２プログラム記述を参照しながら、前記回路仕様を満足する回路のハードウェアを所定のハードウェア記述言語で記述する処理を更に含むことを特徴とする請求項１８記載の論理回路の設計方法。

【請求項２０】　前記第１プログラム記述に対して０サイクルで実行されるループが存在するか否かが判定され、存在しないと判別されたときに前記回路仕様を満足する回路のハードウェアを所定のハードウェア記述言語で記述する処理を行うことを特徴とする請求項１４記載の論理回路の設計方法。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明はプログラム記述からシミュレーション用のプログラム記述又はハードウェアを特定する回路記述を自動生成する技術に関し、例えばパイプライン動作される論理回路、例えばＣＰＵ（Central Processing Unit）等の論理回路の設計に適用して有効な技術に関する。

【０００２】

【従来の技術】

プログラム言語を用いてディジタル回路の回路記述を生成する技術がある。特許文献１に記載の技術では、レジスタを示す変数と、レジスタの入力を示す変数とに分け、モジュール部での処理の後に第２の変数から第１の変数に一括して代入する一括代入部を設けている。特許文献２には、汎用プログラム言語で回路動作を記述したプログラムの中から、順次制御する部分を特定処理部で特定し、その後、変換処理部で、前記順次制御する部分の記述を、ステートマシンとして動作するように汎用プログラム言語を用いて変換し、その変換後のプログラムを取得し、続いて、プログラム生成処理部で、前記変換後のプログラムの中から並行動作する部分を抽出し、この抽出部分の全てをアクセスするプログラムを生成する、というものである。

【０００３】

【特許文献１】

特開２００２－４９６５２号公報

【特許文献２】

特開平１０－１４９３８２号公報

【０００４】

【発明が解決しようとする課題】

特許文献１によれば、①回路動作を示すモジュール、②レジスタ代入を行う一括代入部、③クロック同期で繰り返すループ部の３つの構成からなっており、特に③内で①の実行後に②を実行する事を特徴としている。しかしながら、①がクロック境界を含まず、必ず③に含まれる構成となるため、複数サイクルにまたがる回路動作を記述する為には、回路動作をクロック境界で分割する必要がある。例えば、ある条件が成立したときは、前サイクルで実行した回路動作の途中から回路動作を行うという記述を行わなければならないが、そのような記述を行うのは困難である。特に、ストール動作を伴うパイプライン動作を行う回路を特許文献１に示す方法で記述すると、煩雑な作業を伴い、かつプログラム記述が複雑なものになる虞のあることが本発明者によって見出された。

【０００５】

特許文献２によれば、①汎用言語で記述したプログラムから順次処理部を識別し、ステートマシンを表す汎用プログラム記述に変換、②関数レベルでの並列性の抽出、③ハード化するプログラムとそれを制御するソフトプログラムの結合の自動化、④順次処理部内でハード化する際にフリップフロップやラッチを必要とする部分を識別してＨＤＬに変換、の４つを特徴点がある。しかしながら、クロック境界を明示的に与えられる手段がなく、サイクル精度での記述を直接行う事が出来ない。特許文献２の実施例によれば、クロック境界は関数から関数への間であり、例えばある条件が成立したときは、前サイクルで実行した回路動作の途中から回路動作を行うという記述を行うのが困難である。特に、ストール動作を伴うパイプライン動作を行う回路を特許文献２に示す方法で記述することは可能であるが、煩雑な作業を伴い、かつプログラム記述が複雑になる虞のあることが本発明者によって見出された。

【０００６】

本発明の目的は、クロック境界を明示的に記述したプログラム記述からハードウェア記述を自動生成することができるコンパイラを提供することにある。

【０００７】

本発明の別の目的は、ストール動作を伴うパイプライン動作が可能な回路のプログラム記述又は回路記述を容易に得る事ができるコンパイラを提供することにある。

【０００８】

本発明の更に別の目的は、ストール動作を伴うパイプライン動作が可能な回路の設計を行うことができる論理回路の設計方法を提供することにある。

【０００９】

本発明の前記並びにその他の目的と新規な特徴は本明細書の記述及び添付図面から明らかになるであろう。

【００１０】

【課題を解決するための手段】

本願において開示される発明のうち代表的なものの概要を簡単に説明すれば下記の通りである。

【００１１】

〔１〕本発明の概要を全体的に説明する。即ち、クロック境界（記述子＄）及びレジスタ代入文（演算子＝＄を挟む記述）によりステートメントレベルでの並列動作の記述をサイクル精度で記述可能な擬似Ｃ記述（１）を入力とし、レジスタ代入文の識別を行い（Ｓ２）、実行可能なＣ記述（３）を生成するする（Ｓ３およびＳ４）と共に、状態数削減を行ったステートマシンを抽出し、０サイクルで実行されるループが存在するか否かを判定し（Ｓ５）、もしなければ、論理合成可能な回路記述（４）を生成する（Ｓ６）。

【００１２】

上記より、クロック境界を明示的にＣ記述内に挿入した擬似Ｃ記述を入力し、レジスタ代入文によるステートメントレベルでの並列記述を可能にした擬似Ｃ記述を入力するから、ストール動作を伴うパイプライン動作が表現可能である。

【００１３】

擬似Ｃ記述から一般のＣコンパイラによるコンパイルが可能なＣ記述を出力することができる。状態（ステート）数削減を行うので、記述で与えたクロック境界の数＋１以下のステート数のステートマシンを伴う回路記述を出力することができる。

【００１４】

ステートマシンを意識する事なくプログラム・レベルで機能設計を行う事ができるため、記述量が低減され、開発期間の短縮のみならず品質向上にも寄与する。

【００１５】

また、一般のクロック境界を指定しないプログラム・レベルでの記述では表現できない、バス・インターフェース回路や調停回路の記述が可能となる。特に、レジスタ代入が記述可能である為、ステートメントレベルでの並列性を考慮した記述を行う事が可能であり、ストール動作を伴うパイプライン動作のような複雑な回路動作をＣ記述よりも少ないコード量で容易に記述可能である。

【００１６】

また、一般のＣコンパイラでコンパイル可能なＣ記述へ変換する為、高速なシミュレーションが可能となり、機能検証工数の大幅な低減が可能となる。従って、機能設計における論理設計、論理検証の双方の大幅な工数削減が可能となる。

【００１７】

クロック境界を指定したプログラム記述からミーリー（Ｍｅａｌｙ）型のステートマシンが生成可能であるので、プログラム・レベルでのモデル検査を行う事が可能である。

【００１８】

高位合成ツールが不得意とする、サイクル精度を要求される例えば、キャッシュ・コントローラやＤＭＡコントローラの開発に適用可能であり、設計期間の短縮に大きく寄与する。

【００１９】

〔２〕本発明に係るコンパイラの第１形態では、コンパイラは、所定のプログ

ラム言語を流用して記述された第１プログラム記述（１）を回路記述（４）に変換可能であって、前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文（演算子＝＄を挟む記述）とクロック境界記述（＄）を含み、前記回路記述は、前記第１プログラム記述が特定する回路動作を実現するハードウェアを所定のハードウェア記述言語で特定する。

【００２０】

本発明に係るコンパイラの第２形態では、コンパイラは、所定のプログラム言語を流用して記述された第１プログラム記述を所定のプログラム言語を用いた第２プログラム記述（３）に変換可能であり、前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文（演算子＝＄を挟む記述）とクロック境界記述（＄）を含む。前記第２プログラム記述は、前のサイクルの状態を参照可能に前記レジスタ代入文を変形した変形代入文（１３）と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文（１２）とを含む。

【００２１】

本発明に係るコンパイラの第３形態では、コンパイラは、所定のプログラム言語を流用して記述された第１プログラム記述（１）を、所定のプログラム言語を用いた第２プログラム記述（３）と回路記述（４）に変換可能である。前記第１プログラム記述は、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を含む。前記第２プログラム記述は、前のサイクルの状態を参照可能に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述とを含む。前記回路記述は、前記第２プログラム記述で定義されるハードウェアを所定のハードウェア記述言語で特定する。

【００２２】

前記所定のプログラム言語は例えばＣ言語である。前記ハードウェア記述言語は例えばＲＴＬレベルの記述言語である。

【００２３】

〔３〕本発明に係る論理回路の設計方法の第１形態では、タイミング仕様に基

づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文（演算子＝＄を挟む記述）とクロック境界記述（＄）を含む第１プログラム記述（１）を入力する第１処理（Ｓ１）と、前記第１プログラム記述に基づいて前記タイミング仕様を満足する回路情報を生成する第２処理と、を含む。

【００２４】

前記第２処理は、前記第１プログラム記述を変換して、前記レジスタ代入文が入力変数と出力変数を用いて変形される（Ｓ２）と共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する（Ｓ４）記述（１３，１２）を含む第２プログラム記述（３）を、前記回路情報として生成する処理を含んでよい。

【００２５】

前記第２処理は、前記第２プログラム記述に基づいて前記タイミング仕様を満足するハードウェアを所定のハードウェア記述言語で特定するための回路記述（４）を更に別の前記回路情報として生成する処理を含んでよい。

【００２６】

前記第２プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含んでもよい。

【００２７】

上記第２処理に関し、前記レジスタ代入文が入力変数と出力変数を用いて変形される（Ｓ２）記述（１３）を含む第２プログラム記述（５）と、前記クロック境界記述に対応させて前記入力変数を出力変数に代入する（Ｓ４）記述（１２）を含む第３プログラム記述（３）とを、分けて把握することも可能である。このとき、第３処理によりシミュレーションは第３プログラム記述に基づいて行うことになる。

【００２８】

〔４〕本発明に係る論理回路の設計方法の第２形態では、タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とするレジスタ代入文とクロック境界記述を

含む第１プログラム記述を入力する入力処理（Ｓ１）と、前記レジスタ代入文が入力変数と出力変数を用いて変形される（Ｓ２）と共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する（Ｓ４）記述（１３，１２）を含み、前記所定のプログラム言語で記述された第２プログラム記述を生成する変換処理とを含む。

【００２９】

前記変換処理は、第１プログラム記述に基づいてＣＦＧを生成する過程で、前記ＣＦＧに前記クロック境界記述に対応してクロック境界ノードを設定し、前記クロック境界ノードの後に、前記レジスタ代入記述を挿入する処理であってよい。

【００３０】

第２プログラム記述に対してそのＣＦＧを利用しながらステート遷移毎の変数表を作成しながらコード最適化を行う最適化処理を更に含んでもよい。

【００３１】

前記変数表においてステート間で変数に変化のない部分を前置保持を要する部分として抽出し、抽出された部分に、出力変数に入力変数を代入する代入記述を追加する前置保持処理を更に含んでもよい。

【００３２】

前記前置保持処理を経た変数表の各ステート遷移毎の変数と引数に基づいてステートマシンを構成するコードの抽出を行う抽出処理を更に含んでもよい。

【００３３】

前記抽出処理で抽出されたステートマシン構成コードと第２プログラム記述を参照しながら、前記回路仕様を満足する回路のハードウェアを所定のハードウェア記述言語で記述する回路記述を生成する処理を更に含んでもよい。

【００３４】

前記第１プログラム記述に対して０サイクルで実行されるループが存在するか否かが判定され、存在しないと判別されたときに前記変換処理が行なわれる。

【００３５】

【発明の実施の形態】

《設計方法の概略》

図１には本発明に係る論理回路の設計方法が例示される。同図に示される設計方法は、擬似Ｃ記述（擬似Ｃプログラム）１の作成、擬似Ｃプログラム１に対するコンパイル処理２に大別される。コンパイル処理２では、擬似Ｃプログラム１を、レジスタ代入記述を変形代入文とした擬似Ｃプログラム（５に格納）、および実行可能なＣ記述（Ｃプログラム）３に変換し、また、そのＣプログラム３をＲＴＬ（Register Transfer Level）などのＨＤＬ（Hardware Description Language）記述４に変換する。

【００３６】

前記擬似Ｃプログラム１は、サイクル精度で回路動作を特定可能とするクロック境界記述（単にクロック境界とも記す）及びレジスタ代入文を含み、ステートメントレベルでの並列記述を可能にしたプログラムである。擬似Ｃ記述とは、前記クロック境界及びレジスタ代入文が定義されているない所謂ネイティブのＣ言語記述とは相違するという意味で用いられている。プログラム言語としてＣ言語以外の高級言語をベースとすることを妨げるものではない。

【００３７】

コンパイル処理２は、図示を省略するコンピュータ装置がコンパイラを実行し、擬似Ｃプログラム１を読み込んで行なわれる。先ず擬似Ｃプログラム１が読み込まれる（Ｓ１）。読み込まれた擬似Ｃプログラム１に対しては、レジスタ代入文の識別が行なわれ、識別されたレジスタ代入文を、前のサイクルの状態を参照可能に変形し、換言すれば、入力変数と出力変数を用いて変形する（Ｓ２）。変形されたレジスタ代入文を変形代入文とも称する。レジスタ代入文が変形代入文に変形された擬似Ｃプログラムはレジスタ情報記憶部５に格納される。レジスタ代入文が変形代入文に変形された擬似Ｃプログラムは前記レジスタ情報記憶部５から取り出されて、そのコントロール・フロー・グラフ（以下ＣＦＧと記す）が生成される（Ｓ３）。生成されたＣＦＧは中間表現記憶部６に格納される。前記中間表現記憶部６に格納されたＣＦＧ及び前記レジスタ情報記憶部５に格納された擬似Ｃプログラムは、実行可能なＣ記述プログラムに変換される（Ｓ４）。例えば、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に

伴うレジスタの変化に対応させるレジスタ代入記述挿入文が挿入される。換言すれば、クロック境界記述に対応させて前記変形代入文の入力変数を出力変数に代入するレジスタ代入記述挿入文が挿入される。

【００３８】

前記擬似Ｃプログラム５等に基づいてＨＤＬ記述４を得る場合、先ずそれらを入力してステートマシンの生成が行なわれる（Ｓ５）。ステートマシン生成（Ｓ５）は、ステート数削減処理（Ｓ５Ａ）、コードの最適化（Ｓ５Ｂ）、ＨＤＬ記述に則するための前置保持解析（Ｓ５Ｃ）、及びステートマシン抽出（Ｓ５Ｄ）に大別される。ステート数削減処理（Ｓ５Ａ）とコードの最適化（Ｓ５Ｂ）は最適化処理の範疇に属する処理と把握してもよい。コードの最適化（Ｓ５Ｂ）の段階では、０サイクルで実行されるループが存在するか否かを判定し、もしなければ、ＨＤＬ記述に則するための前置保持解析（Ｓ５Ｃ）、及びステートマシン抽出（Ｓ５Ｄ）が行われる。前記Ｃ記述プログラムを得るときには、例えばクロック境界ノードに前記レジスタ代入記述挿入文を挿入すればよかったが、ＨＤＬ記述を得るときはクロック境界でレジスタ値が変化しない場合にもそれを明示的に記述しておくことが必要とされる。そのために、前置保持解析（Ｓ５Ｃ）が行なわれる。生成されたステートマシンはステート遷移毎の変数表に基づいて生成される。生成されたステートマシンはステートマシン記憶部７に保持される。保持されたステートマシン等に基づいてＨＤＬ記述４が生成される（Ｓ６）。

【００３９】

ＨＤＬ記述４は論理合成ツールを利用することによって論理回路図データに変換可能にされる。前記Ｃ記述３は前記論理合成される論理回路のシミュレーションなどに利用される。

【００４０】

以下に、上記擬似Ｃプログラムとそのコンパイル処理を詳細に説明する。以下の詳細説明は図２の回路に図３の仕様を満足させる回路の設計を一例とする。

【００４１】

《設計対象回路》

図２には図１の設計方法を適用して設計すべき回路例が示される。設計対象回

路１０はストール動作を伴うパイプライン加算回路である。その動作仕様は以下の通りである。

（１）入力信号valid_aが立ち上がると、信号レベルのハイレベルとなったサイクルの入力信号aの値を取り込む。ここではvalid_aが立ち上がり変化を問題にする。

（２）入力信号valid_aの立ち上がりの次サイクル以降で、入力信号valid_bの信号レベルがハイレベルとなると、そのサイクルでの入力信号bの値を取り込む。入力信号valid_bに対してはレベル検出だけで充分とされ、エッジ変化の検出は不要とされる。

（３）上記（１）（２）の動作でaとbが取り込まれたなら、その次サイクルでaとbの加算結果を出力信号outにより送出し、その同一サイクルに出力信号valid_outの信号レベルをハイレベルとし、次サイクルで出力信号valid_outの信号レベルをロウレベルとする。

（４）出力信号outは（１）（２）（３）の動作での新たな加算結果が代入されない限り、同じ値を出力する。

（５）出力信号valid_outは（１）（２）（３）の動作で出力信号outへ新たな加算結果が代入されたサイクルのみ信号レベルがハイとなり、それ以外はロウレベルを出力する。

【００４２】

図３には図２の回路動作仕様を示すタイミングチャートである。同図において、出力データ送出と入力データ取り込みが同一サイクルで行われており、パイプライン動作となっている。例えばa2の入力とa1+b1の出力が並列化されている。また、入力信号valid_aの立ち上がりの次サイクル以降で入力信号valid_bの値が１となった次のサイクルで出力データ送出が行われる為、ストール動作を伴うパイプライン動作となっている。例えばb1の取込み後におけるb2の取込みは2サイクル待たされている。

【００４３】

《擬似Ｃプログラム》

図４には前記設計対象回路１０の擬似Ｃプログラムが例示される。図４に記述

において１１は、設計対象回路１０の回路動作を記述した回路動作記述部である。同図に示される擬似Ｃプログラムの記述は以下の通りである。即ち、

１行目：Ｃ言語でのライブラリ呼び出し、

２～７行目：関数pipelineのプロトタイプ宣言部、

８～１４行目： main関数部、

９～１０行目： main関数のローカル変数宣言部。出力信号はポインタ型で宣言、

１１～１２行目：main関数のローカル変数の初期化（出力信号のみ初期化、特に出力信号に対してＲＴＬへの変換時にレジスタが推定される場合ここで指定した初期値がリセット値となる）、

１５～３６行目：pipeline関数部、

１８～２０行目：pipeline関数のローカル変数宣言部（特にローカル変数に対してＲＴＬへの変換時にレジスタが推定される場合ここで指定した初期値がリセット値となる）、

２１～３５行目：回路動作記述部１１、である。

【００４４】

回路動作記述部１１の詳細は以下の通りである。即ち、

２１、３５行目：無限ループにより回路を表現、

２２行目：入力変数valid_aのローカル変数valid_a_tmpへのレジスタ代入文（ここで、0x0001&valid_aにより、入力変数valid_aの有効ビット幅が１ビットである事を指定している）、

２３行目：valid_aが1 'b1でvalid_a_tmpが1' b0であるか否かの判定文（即ち、valid_aが立ち上がりであるか否かの判定文。特に、0x0001&valid_a_tmpにより、ローカル変数valid_a_tmpの有効ビット幅が１ビットである事を指定している）、

２４行目：入力信号aのローカル変数a_tmpへの代入文（特に、0x7FFF&aにより、入力変数aの有効ビット幅が１５ビットである事を指定している）、

２５行目：クロック境界、

２６行目：gotoラベル、

２７～２８行目：入力変数valid_bが1 'blであれば、ローカル変数b_tmpに入力変数bを代入し、そうでなければクロック境界を１つまたいでラベルＬへ分岐する事を表している（特に、0x0001&valid_bにより、入力変数bの有効ビット幅が１ビットである事を、0x7FFF&bにより、入力変数bの有効ビット幅が１５ビットである事を表している）、

２９行目：ローカル変数a_tmpとローカル変数b_tmpの和の出力変数outへのレジスタ代入文、

３０行目：定数0x0001の出力変数valid_outへのレジスタ代入文、

３１行目：２３行目のif文の判定が成立しなかった場合の分岐。即ち、valid_aが立ち上がりでなかった場合の分岐を表す、

３２行目：クロック境界、

３３行目：定数0x0000の出力信号valid_outへのレジスタ代入文、である。

【００４５】

上記記号“＄”はクロック境界記述を意味し、記号“＝＄”レジスタ代入を意味する。それらはＣ言語の汎用的な記述子及び演算子ではない。これを用いた擬似Ｃプログラムは、その意味においてＣ言語を流用したプログラム記述と言うことができる。

【００４６】

上記回路動作記述部１１より明らかなように、クロック境界記述及びレジスタ代入文によりステートメントレベルで並列動作をサイクル精度で簡単に記述可能になる。サイクル精度とは、クロックサイクルとの同期が意図される、ということである。

【００４７】

図４の回路動作記述部１１の記述内容について説明する。入力変数valid_aをローカル変数valid_a_tmpに代入する事で、if文によるvalid_aの立ち上がり判定を行い、もし立ち上がりであった場合は、ローカル変数a_tmpに入力信号aを取り込み、次のサイクルで入力信号valid_bが1 'blであるか否かを判定する。もしそうなら入力信号bの値をローカル変数b_tmpに代入し、そうでなければ次のサイクルでもう一度入力信号valid_bが1 'blであるか否かを判定する。これを入力信号

valid_bが1 ‘blとなるまで繰り返す。この動作がストール動作に対応している。さて、ローカル変数a_tmpとb_tmpの和は取り込んだaとbの値の和を表しており、それを出力変数outへレジスタ代入し、同時に1’ blを出力信号valid_outへレジスタ代入している。これにより、入力信号aとbを取り込んだ１サイクル後での加算結果とvalid_out信号が1 ‘blである事を表現している。if文によるvalid_aの立ち上がり判定を行い、立ち上がりでない場合は、１サイクル後に1 ‘b0をvalid_outへレジスタ代入している。valid_aの立ち上がりは高々２サイクルに１回しか起こり得ないので、変数outへの新たな代入が２９行目で行われた時のみvalid_outが1 ‘blとなり、それ以外の場合は、1’ b0となる。

【００４８】

図４の第２２行におけるレジスタ代入文は、サイクル精度で動作を特定するのに順序回路としてのレジスタを想定しており、左辺（valid_a_tmp）はレジスタの出力、即ち前サイクルの値を保持している変数として把握可能である。レジスタ代入文の右辺（0x0001&valid_a）は現時点のレジスタ入力として把握可能である。また、図４の第２９行目及び第３０行目に記載のレジスタ代入文に関しては、その後の第３２行におけるクロック境界記述でクロックが消費されるようになっているが、図２及び図３の回路仕様ではその次サイクルでoutを出力するとあり、結果として、out、valid_outに関しては必然的にサイクル精度の記述が必要になるため、それらの記述にはレジスタ代入文が用いられている。

【００４９】

《レジスタ代入文識別》

次にレジスタ代入文識別処理Ｓ２について説明する。前記レジスタ代入文識別処理部では、代入文であって、＝と右辺の間に＄が付加された文を識別し、回路動作記述部１１内のレジスタ代入文、レジスタ代入文の左辺の変数の型と初期値を記憶し、識別したレジスタ代入文signal_latched = $ signal;を

signal_latched_i = signal;

signal_latched = signal_latched_o;

の記述に変更する。signal_latched_iは現時点の入力が与えられる入力変数、signal_latched_oは1サイクル前の出力が当てられる出力変数として把握すること

が可能である。変数宣言部に変更により生じた新たな変数
signal_latched_i, signal_latched_o
を先に記憶しておいた変数の型と初期値を参照して追加する。例えば、
unsigned char　signal_latched = 0x01;
の場合は、
unsigned char　signal_latched_o = 0x01;
unsigned char　signal_latched_i;
を追加する。特に、レジスタ代入の左辺の変数が、ポインタ型の場合（記号*が付されている）は、そのポインタ型を用いて変数宣言を行う。例えば、
unsigned char　*signal_latched;
の場合は、
unsigned char　signal_latched_o = 0x01;
unsigned char　signal_latched_i;
を追加する。特に、追加対象となった変数に対して、同じ型で初期値を０としたフラグ変数も追加予定として、記憶する。この例の場合、
unsigned char　flg_signal_latched = 0x00;
を追加予定変数として記憶する。尚、上記変更を行った記述も記憶する。また、変数の初期値はＨＤＬ変換時に該変数へのレジスタ推定が行われた場合、リセット時の値として用いる。

【００５０】

図５にはレジスタ代入文識別処理Ｓ２によって得られる結果が例示される。図４の擬似Ｃプログラム対して追加変数宣言の記述と変形代入文（レジスタ代入文書き換え）１３の記述が変更されている。

【００５１】

《ＣＦＧ生成》

次にＣＦＧ生成処理について説明する。ＣＦＧとは、一般に各関数内部において制御の流れを示すグラフを意味する。

【００５２】

ＣＦＧ生成処理では、回路動作記述部１１を読み込んで、ＣＦＧの作成を行う

。特にwhileやfor等のループ及びifやcase等の条件分岐、goto文によるラベルへのラベル分岐を識別する為のノードを持つＣＦＧの作成を行う。要するに、whileやfor等のループ及びifやcase等の条件分岐、goto文によるラベルへのラベル分岐をノードに持つＣＦＧを作成する。各文をプログラムの終了迄読み込み、以下の手順１）～７）でノードを作成しながらプログラムの流れに沿って、ノード間の接続を有向辺（向きが付いている辺）で接続する事でＣＦＧを作成する。図６から図２１には手順１）～７）によるＣＦＧの作成過程が順を追って示される。各図にはループ文スタック、分岐文スタック、生成途中のＣＦＧが示される。

【００５３】

１）ループの開始であれば、ループ文スタックにその行番号とwhileやfor等のループを表す終端記号を登録し、ループ開始ノード（ＮＤｓ）を作成し、行番号と終端記号をノードに付加する。また、forやwhileループ終了条件があれば、その条件を適当な記号に代入し、出力枝に付加し、付加した条件を割り当てた記号との対で記憶する。

【００５４】

２）ループの終了であれば、ループ文スタックから先頭にある情報を取り去り、ループ終了を表すループ終了ノードを作成し、行番号と“end of 終端記号”をノードに付加する。但し、continueやbreakはループの終了としては扱わない。また、do-whileループ終了条件があれば、その条件を適当な記号に代入し、出力枝に付加し、付加した条件を割り当てた記号との対で記憶する。

【００５５】

３）条件分岐の開始であれば、分岐文スタックにその行番号とifやcase等の分岐を表す終端記号を登録し、条件分岐開始ノードを作成し、行番号と終端記号をノードに付加する。また、分岐条件を適当な記号に代入し、出力枝に付加し、付加した条件を割り当てた記号との対で記憶する。

【００５６】

４）条件分岐の終了であれば、分岐文スタックから先頭にある情報を取り去り、条件分岐終了を表す条件分岐終了ノードを作成し、行番号と“end of 終端記号”をノードに付加する。

【００５７】

５）ラベルであれば、ラベルを表すラベルノードを作成し、行番号とラベル記号をノードに付加する。

【００５８】

６）クロック境界であれば、クロック境界ノードを作成し、行番号と＄をノードに付加する。

【００５９】

７）上記以外であれば、行番号と文を付加したノードを作成し、１）～６）の何れかに出会うまでノードをマージする。

【００６０】

上記手順によるＣＦＧが作成されるが、以下の説明では、その説明を簡単化するために、図２２に例示されるように、クロック境界や分岐の始点・終点、及びループの始点・終点の情報を付加していないＣＦＧを用いて説明を行う。特に、クロック境界ノードのみ黒丸で、それ以外のループ、条件分岐、ラベル分岐ノードを白丸で表現する。

【００６１】

《Ｃ記述生成》

前記Ｃ記述生成処理Ｓ４について説明する。Ｃ記述生成処理Ｓ４では、前記レジスタ代入文識別処理で追加予定変数として記憶しておいた変数で、レジスタ代入部識別処理で変更した部分（変形代入文）の直下に対応するフラグ変数に１を代入する文の挿入を行い、レジスタ代入文の左辺の変数への代入文でレジスタ代入文でない代入文の直下に対応するフラグ変数に０を代入する文の挿入を行う。また同時に、ローカル変数宣言部に、レジスタ代入文識別部で記憶しておいた変数宣言を追加する。図２３ではflg_valid_a_tmp=1、flg_valid_out=1のフラグが挿入されている。

【００６２】

次にレジスタ代入記述挿入文が決定される。前記レジスタ代入文識別処理Ｓ２において、識別されたレジスタ代入文の右辺の変数全てに対して、レジスタ代入記述挿入文が作成される。即ち、

レジスタ代入文：

signal_latched = $ signal;

変更後の記述：

signal_lacthed_i = signal;

signal_latched = signal_latched_o;

追加された変数：

signal_latched_i, signal_latched_o, flg_signal_latched

とされている場合、下記記述

signal_latched_o = signal_latched_i;

if (flg_signal_latched==1) signal_latched = signal_latched_o;

を作成する。これをレジスタ代入文識処理で識別したレジスタ代入文の右辺の変数全てに対して作成する。例の場合には、下記記述

valid_a_tmp_o = valid_a_tmp_i;

if (flg_valid_a_tmp==1) valid_a_tmp = valid_a_tmp_o;

out_o = out_i;

if (flg_out==1) *out = out_o;

valid_out_o = valid_out_i;

if (flg_valid_out==1) *valid_out = valid_out_o;

が得られる。

【００６３】

上記レジスタ代入記述挿入文は、図２４に例示されるように、クロック境界ノードの直下に挿入される。図２４においてレジスタ代入記述挿入文には参照符号１２が付されている。このようにして行なわれるＣ記述への変換は、各ノードに付加された行番号等の情報を元に、深さ優先探索等のアルゴリズム（ＤＦＳ）を用いて、ＣＦＧを探索する事で挿入文の順番を考慮して行えば良い。尚、適度にコメント文を挿入しても良い。

【００６４】

図２５乃至図２７には上記Ｃ記述生成処理Ｓ４を経て得られる実行可能な変換Ｃ記述（Ｃプログラム）３の全体が例示される。

【００６５】

《ステートマシン生成－ステート数削減》

前記ステートマシンの生成処理Ｓ５について説明する。ステート数削減処理Ｓ５Ａは例えば第１又は第２のルールに従って行なわれる。ステート数削減処理の第１のルールは図２８に例示される。即ち、ループ開始・終了ノード、条件分岐開始・終了ノード、ラベル分岐ノードの何れかであって、入力辺が複数あるノードを探索し、その入力辺の内２つ以上の入力辺にクロック境界がある場合は、同図に示すグラフ変形を行う。ステート数削減処理の第２のルールは図２９に例示される。即ち、ループ開始・終了ノード、条件分岐開始・終了ノード、ラベル分岐ノードの何れかであって、出力辺が複数あり且つ出力辺に付加された条件が入力信号も出力信号の何れも含まず、２本以上の出力辺にクロック境界が付加されたノードを探索し、その前段のクロック境界が出力辺のクロック境界を含まない場合、同図に示すグラフ変形を行う。図３０には図２２のＣＦＧに対してステート数削減を行った結果が例示される。

【００６６】

《ステートマシン生成－コード最適化》

コード最適化処理Ｓ５Ｂでは前記ステート数削減等の処理を行ったＣＦＧに対しては図３１に例示されるようにステートの割り当てを行う。図３１に従えば、回路動作部の開始文に対応するＣＦＧ上のノードに初期ステートを割り当て、ＣＦＧ上のクロック境界ノードにステートを割り当てる。但し、開始ノードへの入力辺が１つしか存在せずクロック境界が付加されている場合は、既に割り当てた初期ステートを削除する。尚、最適化の第１ルールにより、初期ステート削除が起こる必要十分条件は、開始ノードへの入力辺が１つしか存在せずクロック境界が付加されている事である事に注意することが望ましい。また、得られるステート数は、必ず回路動作部に記述したクロック境界の数＋１以下となる事に注意すべきである。

【００６７】

ここで、前記コード最適化の処理を、特別に簡素化した別の例を用いて、図３２乃至図４８を参照しながら説明する。

【００６８】

図３２はコード最適化対象とされる擬似Ｃプログラムを示す。この擬似Ｃプログラムに基づいて得られたＣＦＧは図３３に例示される。図３４には図３３のＣＦＧに対してステート割り当てが行なわれた状態を例示する。

【００６９】

図３５から図４０まではステートマシン生成のための変数表作成処理の様子が順を追って例示される。変数表の作成は、以下の（１）～（３）の手順で行う。（１）ローカル変数を取得し、（２）関数の引数を取得し、（３）割り当てたステートからステートに到達するまでＣＦＧを下位側に辿って、ステート遷移を識別すると共に変数の定義・参照の情報を取得する。この段階で、両端がクロック境界ではないループが発見されると、ゼロサイクル・ループを検出したとして、ユーザに通知し、処理を終了。ゼロサイクル・ループの発生は、生成される回路に組合せ回路からなるループ回路が存在する事を意味しており、ループ回路の存在は生成される回路に重大なミスがあることを意味する。図３５にはステートＳＴ０からＳＴ１への一つのステート遷移におけるローカル変数と引数が例示される。図３６にはステートＳＴ０からＳＴ１への別のステート遷移におけるローカル変数と引数が例示される。図３７にはステートＳＴ０からＳＴ２へのステート遷移におけるローカル変数と引数が例示される。図３８にはステートＳＴ１からＳＴ０へのステート遷移におけるローカル変数と引数が例示される。図３９にはステートＳＴ２からＳＴ０へのステート遷移におけるローカル変数と引数が例示される。図３５から図３９に示される夫々のステート遷移で得られたローカル変数と引数に基づいて、図４０に例示される変数表が生成される。図４０の変数表の記述において、def[n]：ｎ行目で変数定義されている事を表し、
use@var[m]：m行目で変数varへの代入に用いられている事を表し、
pred(cond)｛…｝：条件condの分岐が成立した場合、｛…｝が実施される事を表し、
def[l]use：ｌ行目で自変数への代入に用いられている事を表し、
use@pred(cond)：条件condで用いられている事を表す、とされる。

【００７０】

最適化処理は例えば図４０の変数表に基づいて行なわれる。最適化処理の一つ

は冗長ステートメントの削除である。

【００７１】

冗長ステートメントの削除として、第１に、同一変数に対して、ステート遷移のカラム内でdefが２つ以上存在する場合には、１）又は２）の処理を行う。即ち、

１）下記１－１），１－２）をdefの後段に存在するpred(cond){...}の手前まで(pred(cond){...}の有無に関わらず)実施する。１－１）：defの後段にuseを伴うdefがない場合は、最後のdefに対応するステートメントのみ残す。１－２）：defの後段にuseを伴うdefがある場合は、useを伴うdefの後段にuseを伴わないdefがあれば、そのdefのみを残し、そうでなければuseを伴うdefの前段のdefとuseを伴うdefを残し、これを変化が無くなるまで繰り返し、残ったdefに対応するステートメントのみ残す。

２）defの後段にpred(cond){...}が無ければ終了し、あれば下記２－１），２－２）を実施する。２－１）：pred(cond){...}の条件がdefの結果を参照している場合には終了とする。２－２）：pred(cond){...}の条件がdefの結果を参照していない場合は、１）へ分岐とする。

【００７２】

冗長ステートメントの削除処理として、第２に、useがどのステート遷移にも存在しない変数は削除とする。

【００７３】

上記処理手順により図４０の変数表に対して冗長ステートメント削除を行ったとき、削除されるべきステートメントは図４１に示される。同図において削除されるべきステートメントには斜め破線が明示されている。図４２には冗長ステートメントが削除された結果の変数表が例示される。図４３には冗長ステートメントが削除された結果をＣＦＧで表している。

【００７４】

最適化処理のもう一つはローカル変数の削除である。このローカル変数の削除処理として、第１に、各変数のステート遷移カラムに於いて、下記１）～３）を左から順次変化が無くなる迄実施する。即ち、

１）defの後段にpred(cond){...}を挟まずuseが存在する場合には、１－１）、１－２）、１－３）、１－４）を行う。１－１）：use自体がuse@predの場合は代入操作を実施し、defを削除し、１－２）の場合は@の変数がローカル変数でuse@predとして用いられている場合は代入操作を実施せず、１－３）：@の変数がローカル変数でuse@predとして用いられていない場合は代入操作を実施し、defを削除し、１－４）：@の変数が引数の場合は代入操作を実施し、defを削除する。

２）defの後段にpred(cond){...}を挟んでuseが存在し、pred(cond){...}の条件で用いられている変数が引数の場合は１－１）から１－４）を適用する。

３）defの後段にpred(cond){...}を挟んでuseが存在し、pred(cond){...}の条件で用いられている変数がローカル変数の場合には、３－１）、３－２）を行う。３－１）：pred(cond){...}の条件がdefの結果を参照していない場合は１－１）から１－４）を適用し、３－２）：pred(cond){...}の条件がdefの結果を参照している場合は代入操作を実施しない。

【００７５】

ローカル変数の削除処理として、第２に、defがどのステート遷移カラムにも存在しない変数は削除し、代入操作後のＣＦＧを再び解析して、変数表を更新する。

【００７６】

図４２の変数表に対してローカル変数削除を行ったとき、削除されるべき変数は図４４に示される。同図において削除されるべき変数には斜め破線が明示されている。図４５にはローカル変数削除処理が行なわれた結果をＣＦＧで表している。

【００７７】

図４６には冗長ステートメント削除処理及びローカル変数削除処理が行なわれて最終的に更新された変数表が例示される。この変数表により、必要となるローカル変数が管理されることになる。図４６において、defが存在しない部分では、出力変数とローカル変数の前置保持が必要である事が識別できる。ステートの遷移において当然前置保持されなければならないからである。従って、その部分

には、前置保持を必要とすることが容易に識別可能になる。図４７に例示されるように、後工程の前置保持解析により、その部分に、前置保持“retain“が追加されることになる。

【００７８】

コードの最適化として更に、図４８に例示されるような演算式の簡約化が行なわれる。

【００７９】

《ステートマシン生成ー前置保持解析》

ここからの説明は再度図２及び図３の仕様を満足させる回路設計の例に話しを戻す。図３２乃至図４８では特別に簡素化した別の例を用いて前記コード最適化の処理を説明したが、図３１に示されるステート割り当てが行われた後に、それと同様の最適化処理を施すことにより、図４９の最適化後のＣＦＧを得ることができ、また、図５０の変数表が得られる。最適化処理後の図５０の変数表には前置保持“retain”は明示されていない。次に説明する前置保持解析で取得される。

【００８０】

図５１には前置保持解析のアルゴリズムが例示される。前置保持解析は、出力変数とローカル変数に対して、状態遷移のカラムにてdefが全く存在しない場合、その状態遷移では前置保持が必要となる。また、defが存在したとしても、pred()が付加されている場合は、各出力変数・ローカル変数の各状態遷移に対して、図５１に示されるような図を作成して、pred()による分岐の中でどの部分で前置保持が必要となるかを識別する。特に、レジスタ代入文識別処理で新たに追加したローカル変数に対しては、_iが付加されている変数のみに対して前置保持が必要かの解析を行う。また、例え、変数表にpred()の情報がなくても必要ならＣＦＧを解析し直して付加する。

【００８１】

図５１に示される図の作成は、pred()の条件を分岐として、use、def等のノードを持つ木を作成する事で行う。そして木のdefより下位の部分木を削除し、最上位ノード以外で下位ノードにdefが存在しないノードを前置保持が必要なノー

ドとして識別する。

【００８２】

ここで、最上位ノードとは、木の根から一番距離が近いdefかuseのノードまでのノードとその兄弟ノード全てを指す。

【００８３】

例えば、変数varの状態遷移STn－＞STmでの変数表からの情報が、pred(cond_0){pred(cond_1){use@var_1[j], pred(cond_2){def[k], pred(cond_3){def[s]}}}である場合、図５１のようになる。

【００８４】

図５２には前置保持解析の結果に対応する変数表が例示される。前置保持を要する部分には“retain”が追加される。

【００８５】

変数表において“retain”の部分に追加すべき実際のコードは変数表から取得することができる。即ち、前置保持解析結果の変数表からの情報取得処理では、変数表のカラムでretainが挿入された、出力変数・ローカル変数を取得し、例えば変数名が

１）レジスタ代入文の左辺の変数の場合は、sig = sig_o;

２）レジスタ代入文識別部で追加した変数であって、_i が付加されている変数の場合は、sig_i = sig_o;

３）その他の変数の場合は、nxt_sig = sig;

として記憶しておく。特に、retainにpred()が付加されている場合は、例えば、pred(cond_0){pred(cond_1){pred(!cond_2){retain}}} に対しては、変数が３）の場合で変数名が sig の場合、pred(cond_0){pred(cond_1){pred(!cond_2){nxt_sig = sig}}}として記憶する。以上の情報を変数表に上書き登録する。図５３には“retain”が実際のコードで上書きされた変数表が例示される。更に、nxt_sigといった具合にnxt_を付加した変数を記憶しておく。図５３の例の場合、nxt_を付加した変数は、a_tmpのみである。

【００８６】

《ステートマシン生成－ステートマシン抽出》

次にステートマシン抽出処理Ｓ５Ｄについて説明する。ステートマシン抽出処理Ｓ５Ｄでは、割り当てた各ステートから深さ優先探索でクロック境界即ちステートであって初期ステートでないステートに到達するまで探索し、その探索で得られたループでも条件分岐でもラベル分岐でもないノードの情報を取得し、変数表のretain情報とマージして、ＨＤＬ記述に用いるステートマシンの抽出を行う。例えば図５４には開始ステートＳＴ０の例が示される。ステートの記述は、特に制限されないが、各ステートからＤＦＳでコードを生成する。この場合ステート変数は、nxt_state = ST0;等の形式として、コード生成を行う。

【００８７】

特に、retain情報にてnxt_が付加された変数はもとの変数名ではなくnxt_が付加された変数名を用いてＨＤＬ記述に用いるステートマシンの抽出を行う。また、信号と定数との＆演算はビット幅解析に用いたので、不要となるため削除する。尚、定数は入力左辺のビット数を勘案してＨＤＬの２進表記に変換してＨＤＬ記述に則した記述とする。

【００８８】

変数表のretain情報の取得では、各状態遷移カラムから、深さ優先探索を開始したステートと同じステートを開始ステートするカラムを全て取得し、retainが開始ステートのみに依存するか、到達ステートにも依存するか、または到達ステートと分岐条件に依存するかを識別し、開始ステートにのみ依存する場合以外は、retain情報のpred()とＣＦＧの分岐条件を比較する事で、ＨＤＬコードの適切な位置にretain情報として変数表に格納した代入式を挿入する。図５５にはretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子が例示される。

【００８９】

図５４及び図５５には開始ステートＳＴ０におけるＨＤＬ記述に則したステートマシン記述の取得例が示される。図５６及び図５７の例は開始ステートＳＴ１におけるＨＤＬ記述に則したステートマシン記述の取得例が示される。図５８及び図５９の例は開始ステートＳＴ２におけるＨＤＬ記述に則したステートマシン記述の取得例が示される。

【００９０】

《ＨＤＬ記述生成処理》

ＨＤＬ記述生成処理Ｓ６において、モジュール宣言は、回路動作記述部を含むＣ記述の関数宣言から、型とポインタを表す＊を削除したものにclkとreset_nを加えたものをＨＤＬ記述として生成する。入出力宣言は、前記関数宣言での引数であって、代入式の左辺にのみ存在する変数を出力とし、代入式の右辺にのみ存在する変数を入力とし、ビット幅はＣ記述の記述内容で説明した方法で識別し、ＨＤＬ記述として生成する。reg宣言はＣ記述に記載されていたローカル変数で、これまでの変換仮定で最終的に残った変数と、これまでの変換仮定で追加された変数とを識別し、clkとreset_nのreg宣言文とともにＨＤＬ記述として生成する。ＣＦＧ生成過程で分岐条件に割り当てた変数のwire宣言のＨＤＬ記述を生成し、前記割り当てた変数への分岐条件の代入文をassign文としてＨＤＬ記述を生成する。また、割り当てたステートを２進数で表す為のparameter宣言文のＨＤＬ記述を生成する。

【００９１】

また、レジスタ代入文に関しては、全てのレジスタ代入文とその右辺の変数宣言を取得し、例えば、取得した情報が

```
unsigned char sig1_latched = 0x00;
unsigned char sig2_latched = 0x00;
unsigned short out;
sig1_latched = $ sig1&0x03;
sig2_latched = $ sig2&0x13;
out          = $ exe_result&0x1FFF;
```

の場合、

```
always @ (posedge clk or negedge reset_n) begin
  if (!reset_n) begin
    sig1_latched_o <= 2' b00;
    sig2_latched_o <= 3' b000;
  end
```

```
  else begin
    sig1_latched_o <= sig1_latched_i;
    sig2_latched_o <= sig2_latched_i;
    out_o          <= out_i;
  end
end
```

のようなＨＤＬ記述を生成する。

【００９２】

次いで、ステートマシン抽出部で得たnxt_が付加された変数の記憶を参照し、その変数の宣言部を取得し、例えば、

この例の場合、a_tmpが対象となるが、

```
unsigned short nxt_a_tmp = 0x0000;
```

であり、reg宣言記述生成時に、

```
a_tmp = $ 0x7FFF&a;
```

なる代入から有効ビット幅が１５ビットである事が解っているので、下記

```
always @(posedge clk or reset_n) begin
  if (!reset_n) begin
    state = ST0;
    a_tmp = 15' b000000000000000;
  end
  else begin
    state = nxt_state;
    a_tmp = nxt_a_tmp;
  end
end
```

の記述を生成する。

【００９３】

また、抽出されたステートマシンのＨＤＬ記述をつなげ、各ステートでの代入文の左辺に対して、レジスタ代入の左辺の変数とレジスタ代入文識別部で追加し

た変数には、対応する_oの変数を代入し、それ以外の変数には初期値を代入した文を作成し、nxt_state=ST0;なる文を作成し、case文のdefaultに対応する部分を作成し、それもつなげ、右辺の変数とwire宣言した変数をorで並べ、下記

```
always @ (state or c1 or c2 or valid_a_tmp_i or valid_a_tmp_o or valid_a_tmp or a_tmp or
        valid_out_i or valid_out_o or out_i or out_o) begin
  case(state[1:0])
  endcase
end
```

の記述を生成し、case文の間につなげたＨＤＬ記述を挿入し、最後の行にendmoduleを付加する事でＨＤＬ記述を生成する。行数は付加しただけである。

【００９４】

図６０乃至図６２にはＨＤＬ記述生成処理Ｓ６にて生成されたＨＤＬ記述が例示される。

【００９５】

以上説明した設計方法によれば、以下の作用効果を得る。

【００９６】

クロック境界を明示的にＣ記述内に挿入した擬似Ｃ記述を入力し、レジスタ代入文によるステートメントレベルでの並列記述を可能にした擬似Ｃ記述を入力するから、ストール動作を伴うパイプライン動作が表現可能である。

【００９７】

ステートマシンを意識する事なくプログラム・レベルで機能設計を行う事ができるため、記述量が低減され、開発期間の短縮のみならず品質向上にも寄与する。

【００９８】

また、一般のクロック境界を指定しないプログラム・レベルでの記述では表現できない、バス・インターフェース回路や調停回路の記述が可能となる。特に、レジスタ代入が記述可能である為、ステートメントレベルでの並列性を考慮した記述を行う事が可能であり、ストール動作を伴うパイプライン動作のような複雑

な回路動作をＣ記述よりも少ないコード量で容易に記述可能である。

【００９９】

また、一般のＣコンパイラでコンパイル可能なＣ記述へ変換する為、高速なシミュレーションが可能となり、機能検証工数の大幅な低減が可能となる。従って、機能設計における論理設計、論理検証の双方の大幅な工数削減が可能となる。

【０１００】

高位合成ツールが不得意とする、サイクル精度を要求される例えば、キャッシュ・コントローラやＤＭＡコントローラの開発に適用可能であり、設計期間の短縮に大きく寄与する。

【０１０１】

以上本発明者によってなされた発明を実施形態に基づいて具体的に説明したが、本発明はそれに限定されるものではなく、その要旨を逸脱しない範囲において種々変更可能であることは言うまでもない。

【０１０２】

例えば、以上説明したプログラム記述及び回路記述は一例であり種々の論理設計に適用することができる。ＨＤＬは必ずしもＲＴＬに限定されない。プログラム記述言語はＣ言語に限定されず、その他の高級言語であってもよい。更にＪａｖａ（登録商標）等の仮想マシン言語などを用いることも可能である。

【０１０３】

【発明の効果】

本願において開示される発明のうち代表的なものによって得られる効果を簡単に説明すれば下記の通りである。

【０１０４】

すなわち、本発明に係るコンパイラによれば、クロック境界を明示的に記述したプログラム記述からハードウェア記述を自動生成することができる。

【０１０５】

本発明に係るコンパイラによれば、ストール動作を伴うパイプライン動作が可能な回路のプログラム記述又は回路記述を容易に得る事ができる。

【０１０６】

本発明に係る論理回路の設計方法によれば、ストール動作を伴うパイプライン動作が可能な回路の設計を行うことができる。

【図面の簡単な説明】

【図１】

本発明に係る論理回路の設計方法を例示するフローチャートである。

【図２】

図１の設計方法を適用して設計すべき回路例を示すブロック図である。

【図３】

図２の回路動作仕様を示すタイミングチャートである。

【図４】

図２の設計対象回路の擬似Ｃプログラムを例示する説明図である。

【図５】

レジスタ代入文識別処理（Ｓ２）によって得られる追加変数宣言の記述とレジスタ代入文書き換えの記述を示す説明図である。

【図６】

擬似Ｃ記述に基づくＣＦＧ作成過程の一つの過程を示す説明図である。

【図７】

擬似Ｃ記述に基づくＣＦＧ作成過程の別の過程を示す説明図である。

【図８】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図９】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１０】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１１】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１２】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１３】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１４】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１５】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１６】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１７】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１８】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図１９】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図２０】

擬似Ｃ記述に基づくＣＦＧ作成過程の更に別の過程を示す説明図である。

【図２１】

擬似Ｃ記述に基づくＣＦＧ作成過程の最終過程を示す説明図である。

【図２２】

説明を簡単化するために図２１のＣＦＧに対してクロック境界や分岐の始点・終点、及びループの始点・終点の情報を付加していないＣＦＧを例示する説明図である。

【図２３】

図２２のＣＦＧに対するフラグ挿入状態を例示する説明図である。

【図２４】

レジスタ代入記述挿入文の挿入位置をＣＦＧ上で例示する説明図である。

【図２５】

Ｃ記述生成処理（Ｓ４）を経て得られる実行可能な変換Ｃ記述（Ｃプログラム）の最初の一部を例示する説明図である。

【図２６】

図２５に続く実行可能な変換Ｃ記述（Ｃプログラム）の一部を例示する説明図である。

【図２７】

図２６に続く実行可能な変換Ｃ記述（Ｃプログラム）の最後の部分を例示する説明図である。

【図２８】

ステート数削減処理の第１のルールを示す説明図である。

【図２９】

ステート数削減処理の第２のルールを示す説明図である。

【図３０】

図２２のＣＦＧに対してステート数削減を行った結果を例示する説明図である。

【図３１】

ステート数削減等の処理を行ったＣＦＧに対してステートの割り当てを行った状態を例示する説明図である。

【図３２】

前記コード最適化の処理を説明するために特別に簡素化した例としてコード最適化対象とされる擬似Ｃプログラムを示す説明図である。

【図３３】

図３２の擬似Ｃプログラムに基づいて得られたＣＦＧを例示する説明図である。

【図３４】

図３３のＣＦＧに対してステート割り当てが行なわれた状態を例示する説明図である。

【図３５】

図３４のＣＦＧに対してステートマシン生成のための変数表作成処理過程の最初の状態を例示する説明図である。

【図３６】

図３５に続く変数表作成処理過程の次の状態を例示する説明図である。

【図３７】

図３６に続く変数表作成処理過程の次の状態を例示する説明図である。

【図３８】

図３７に続く変数表作成処理過程の次の状態を例示する説明図である。

【図３９】

図３８に続く変数表作成処理過程の次の状態を例示する説明図である。

【図４０】

図３９の生成過程を経て生成された変数表を例示する説明図である。

【図４１】

図４０の変数表に対して冗長ステートメント削除を行ったとき、削除されるべきステートメントを例示する説明図である。

【図４２】

図４１に対して冗長ステートメントが削除された結果の変数表を例示する説明図である。

【図４３】

冗長ステートメントが削除された結果をＣＦＧで示す説明図である。

【図４４】

図４２の変数表に対してローカル変数削除を行ったとき削除されるべき変数を例示する説明図である。

【図４５】

ローカル変数削除処理が行なわれた結果をＣＦＧで示す説明図である。

【図４６】

冗長ステートメント削除処理及びローカル変数削除処理が行なわれて最終的に更新された変数表を例示する説明図である。

【図４７】

後工程の前置保持解析により変数表に前置保持“retain“の記述が追加された状態を例示する説明図である。

【図４８】

コードの最適化として更に演算式の簡約化を行った例をＣＦＧで示す説明図で

ある。

【図４９】

図３２乃至図４８で特別に簡素化した別の例を用いて説明したコード最適化の処理を図３１に示されるステート割り当てが行われた後に施すことによって得られる最適化後のＣＦＧを例示する説明図である。

【図５０】

図４９に対する最適化処理後の変数表を示す説明図である。

【図５１】

前置保持解析のアルゴリズムを例示する説明図である。

【図５２】

前置保持解析の結果に対応する変数表を示す説明図である。

【図５３】

図５２に対し“retain”を実際のコードで上書きした変数表を示す説明図である。

【図５４】

開始ステートＳＴ０におけるステートマシン抽出処理を示す説明図である。

【図５５】

図５４に対しretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子を示す説明図である。

【図５６】

開始ステートＳＴ１におけるステートマシン抽出処理を示す説明図である。

【図５７】

図５６に対しretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子を示す説明図である。

【図５８】

開始ステートＳＴ２におけるステートマシン抽出処理を示す説明図である。

【図５９】

図５８に対しretain情報に応ずるコードを変数表から抜き出してステートマシンの抽出に利用する様子を示す説明図である。

【図６０】

ＨＤＬ記述生成処理（Ｓ６）にて生成されたＨＤＬ記述の最初の一部を示す説明図である。

【図６１】

図６０に続くＨＤＬ記述の一部を示す説明図である。

【図６２】

図６１に続くＨＤＬ記述の最後の部分を示す説明図である。

【符号の説明】

1　擬似Ｃ記述（擬似Ｃプログラム）

2　コンパイル処理

3　Ｃ記述（Ｃプログラム）

4　ＨＤＬ記述

5　レジスタ情報記憶部

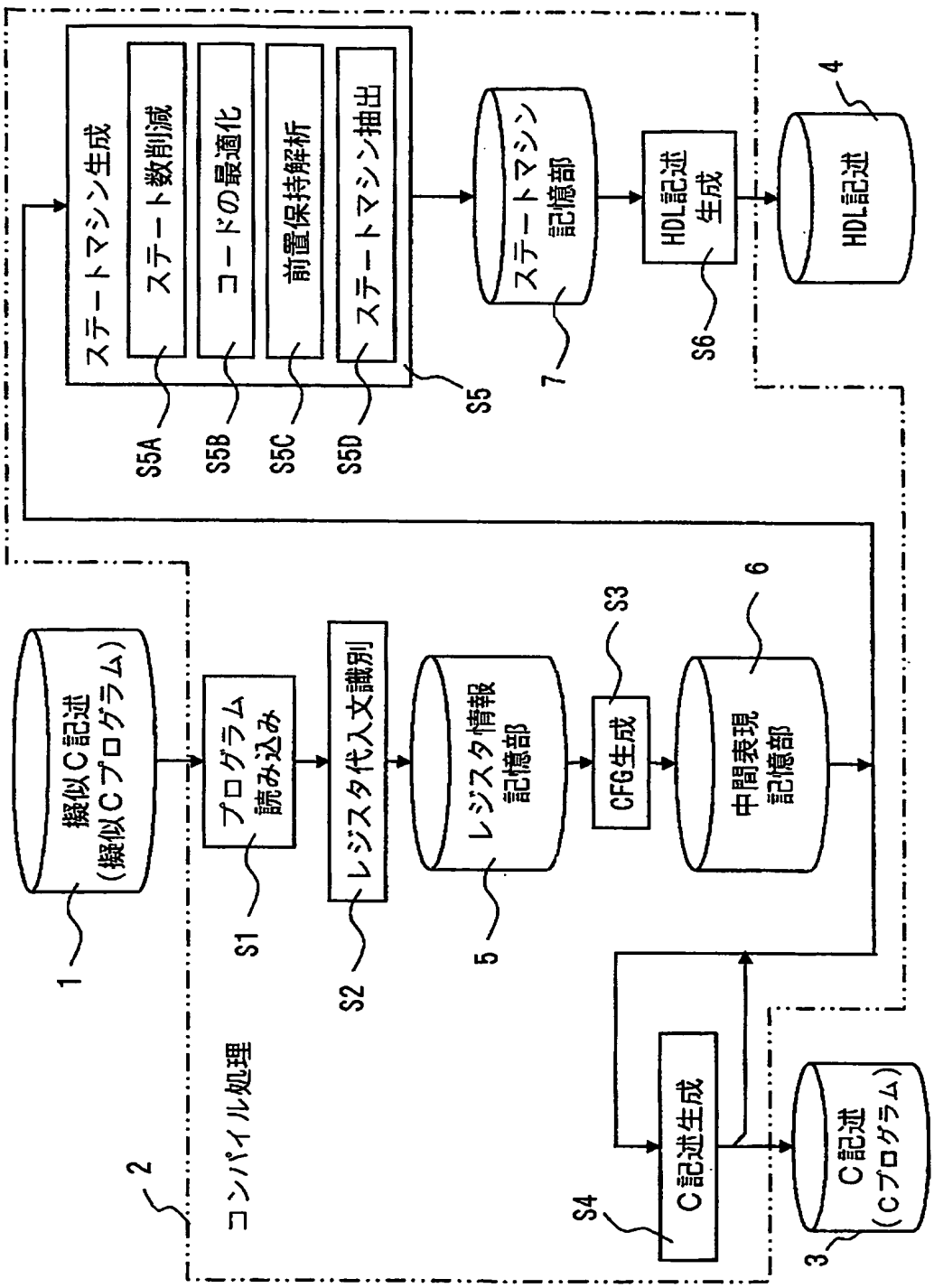
6　中間表現記憶部

7　ステートマシン記憶部

【書類名】　図面

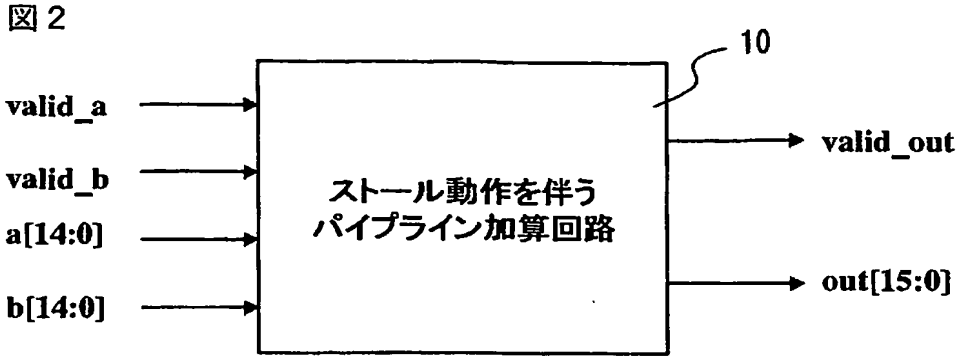
【図１】

図１


擬似C記述
(擬似Cプログラム)
1
プログラム
読み込み
S1
レジスタ代入文識別
S2
レジスタ情報
記憶部
5
CFG生成
S3
中間表現
記憶部
6
C記述生成
S4
C記述
(Cプログラム)
3
コンパイル処理
2
ステートマシン生成
ステート数削減
コードの最適化
前置保持解析
ステートマシン抽出
S5A
S5B
S5C
S5D
S5
ステートマシン
記憶部
7
HDL記述
生成
S6
HDL記述
4

【図２】

図２


10
valid_a
valid_b
a[14:0]
b[14:0]
ストール動作を伴う
パイプライン加算回路
valid_out
out[15:0]

【図３】

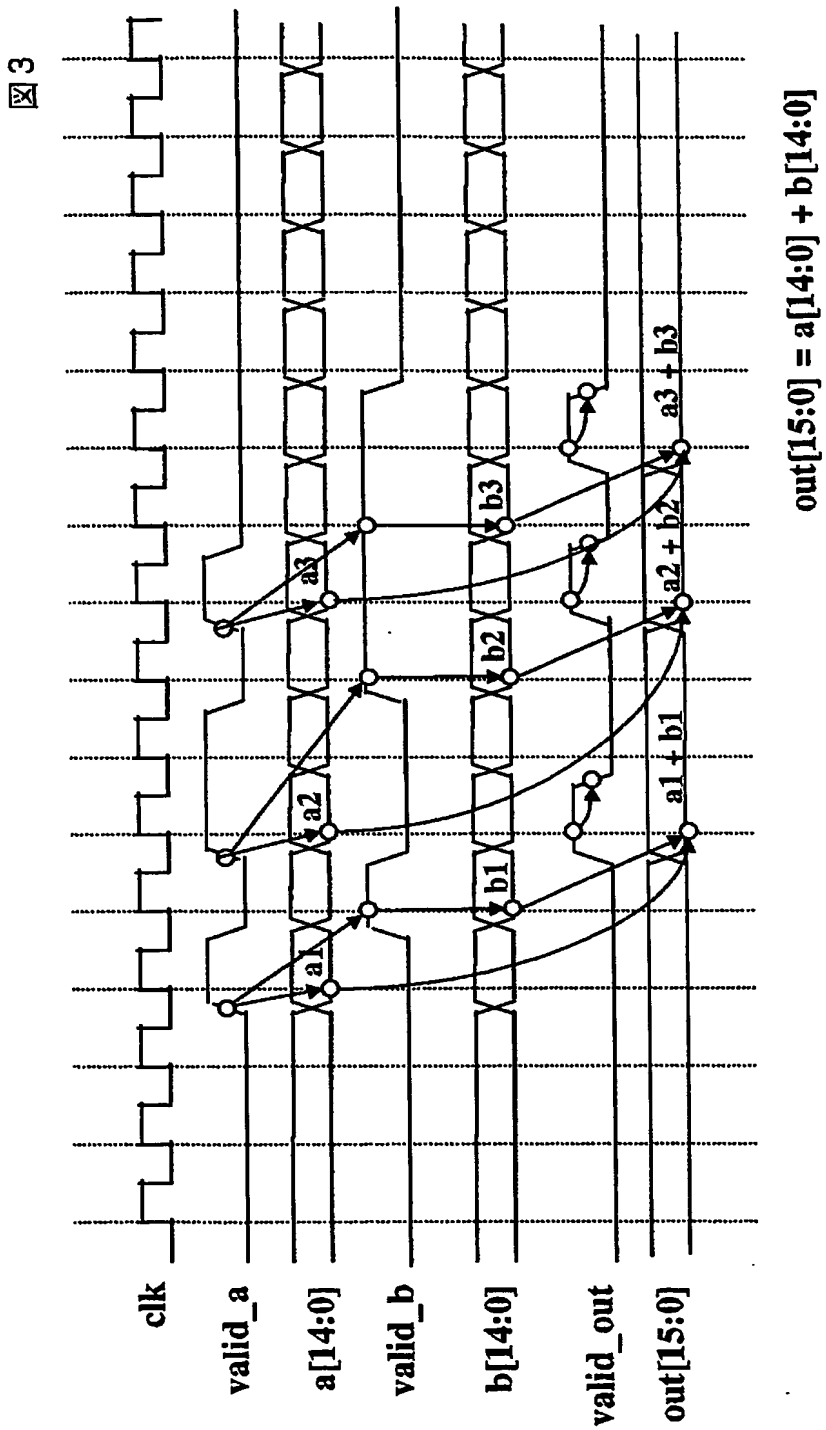


図３
clk
valid_a
a[14:0]
valid_b
b[14:0]
valid_out
out[15:0]
a1
a2
a3
b1
b2
b3
a1 + b1
a2 + b2
a3 + b3
out[15:0] = a[14:0] + b[14:0]

図４

```
1  #include <stdio.h>
2  void pipeline(unsigned short valid_a,
3                unsigned short valid_b,
4                unsigned short a,
5                unsigned short b,
6                unsigned short *out,
7                unsigned short *valid_out);
8  main() {
9    unsigned short valid_a, valid_b,
10                  a, b, *out, *valid_out;
11   *out       = 0x0000;
12   *valid_out = 0x0000;
13   pipeline(valid_a, valid_b, a, b, out, valid_out);
14 }
```

```
15 void pipeline(unsigned short valid_a, unsigned short valid_b,
16               unsigned short a, unsigned short b,
17               unsigned short *out, unsigned short *valid_out) {
18   unsigned short valid_a_tmp = 0x0000;
19   unsigned short a_tmp = 0x0000;
20   unsigned short b_tmp = 0x0000;
21   while (1) {
22     valid_a_tmp = $ 0x0001&valid_a;
23     if ((0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)) {
24       a_tmp = 0x7FFF&a;
25       $
26     L :
27       if (0x0001&valid_b == 0x0001) b_tmp = 0x7FFF&b;
28       else                          $ goto L;
29       *out = $ (a_tmp + b_tmp);
30       *valid_out = $ 0x0001;
31     } else {
32       $
33       *valid_out = $ 0x0000;
34     }
35   }
36 }
```

11(回路動作記述部)

【図５】

図５

```
 1 #include <stdio.h>
 2 void pipeline(unsigned short valid_a,
 3              unsigned short valid_b,
 4              unsigned short a,
 5              unsigned short b,
 6              unsigned short *out,
 7              unsigned short *valid_out);
 8 main() {
 9   unsigned short valid_a, valid_b,
10                  a, b, *out, *valid_out;
11   *out       = 0x0000;
12   *valid_out = 0x0000;
13   pipeline(valid_a, valid_b, a, b, out, valid_out);
14 }

15 void pipeline(unsigned short valid_a, unsigned short valid_b,
16              unsigned short a, unsigned short b,
17              unsigned short *out, unsigned short *valid_out) {
18   unsigned short valid_a_tmp = 0x0000;
19   unsigned short a_tmp = 0x0000;
20   unsigned short b_tmp = 0x0000;
21   unsigned short valid_a_tmp_i;
22   unsigned short valid_a_tmp_o = 0x0000;
23   unsigned short out_i;
24   unsigned short out_o = 0x0000;
25   unsigned short valid_out_i;
26   unsigned short valid_out_o = 0x0000;
27   while (1) {
28     valid_a_tmp_i = 0x0001&valid_a;
29     valid_a_tmp   = valid_a_tmp_o;
30     if ((0x0001&valid_a_tmp == = 0x0000) && (0x0001&valid_a = = 0x0001)) {
31       a_tmp = 0x7FFF&a;
32       $
33     L:
34     if (0x0001&valid_b = = 0x0001) b_tmp = 0x7FFF&b;
35     else                $ goto L;
36     out_i = (a_tmp + b_tmp);
37     *out = out_o;
38     valid_out_i = 0x0001;
39     *valid_out = valid_out_o;
40   } else {
41     $
42     valid_out_i = 0x0001;
43     *valid_out = valid_out_o;
45   }
46 }
47 }
```

追加変数宣言

13(レジスタ代入文書き換え文)

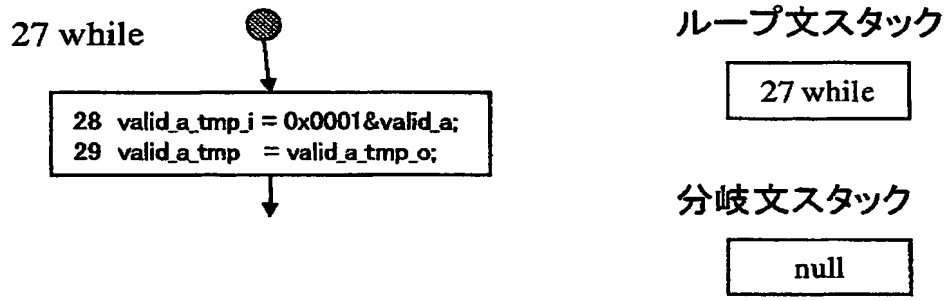
【図６】

図６


27 while
ループ文スタック
27 while
分岐文スタック
null

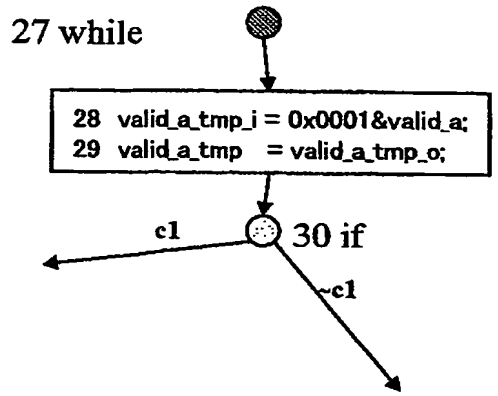
【図７】

図７


27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
ループ文スタック
27 while
分岐文スタック
null

【図８】

図８



ループ文スタック

27 while

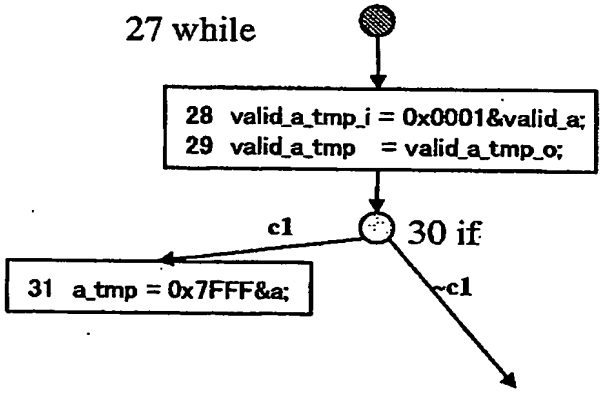
分岐文スタック

30 if

C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

【図９】

図９



ループ文スタック

27 while

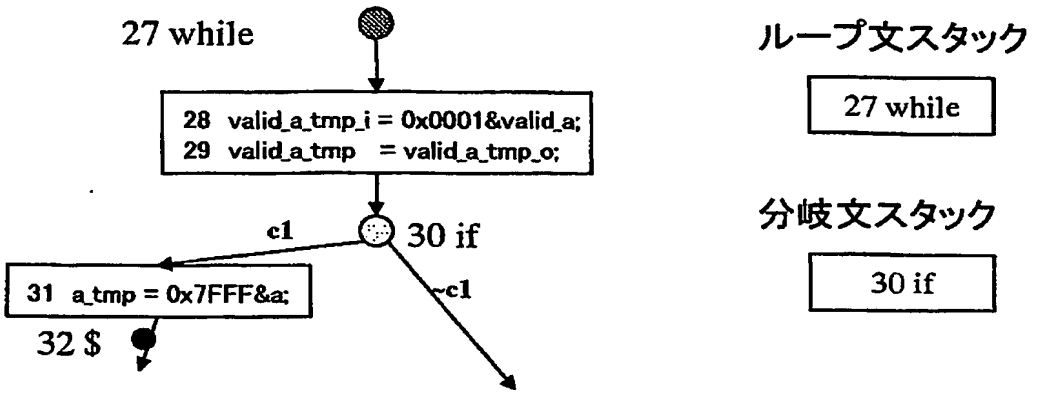
分岐文スタック

30 if

C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

【図１０】

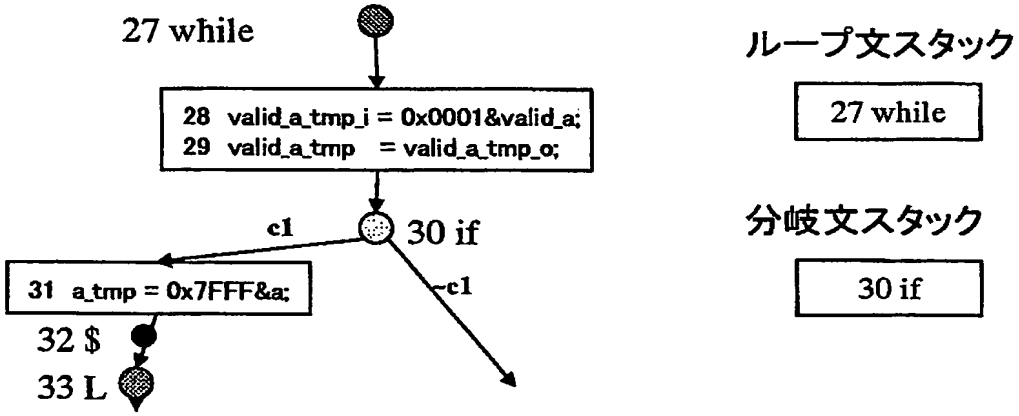
図１０



C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

【図１１】

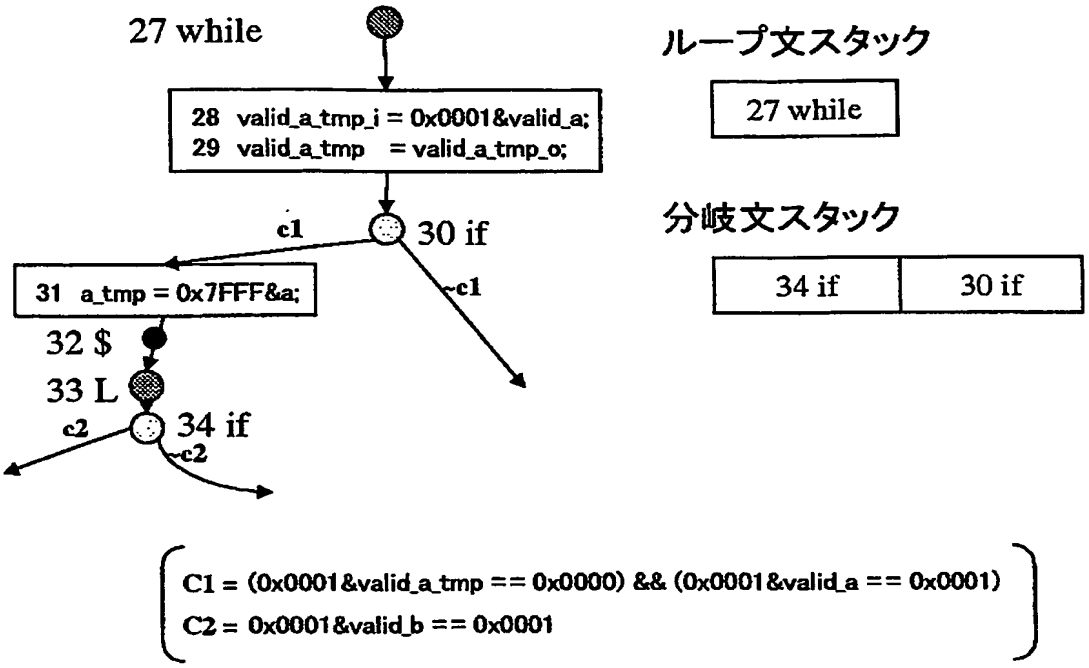
図１１



C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)

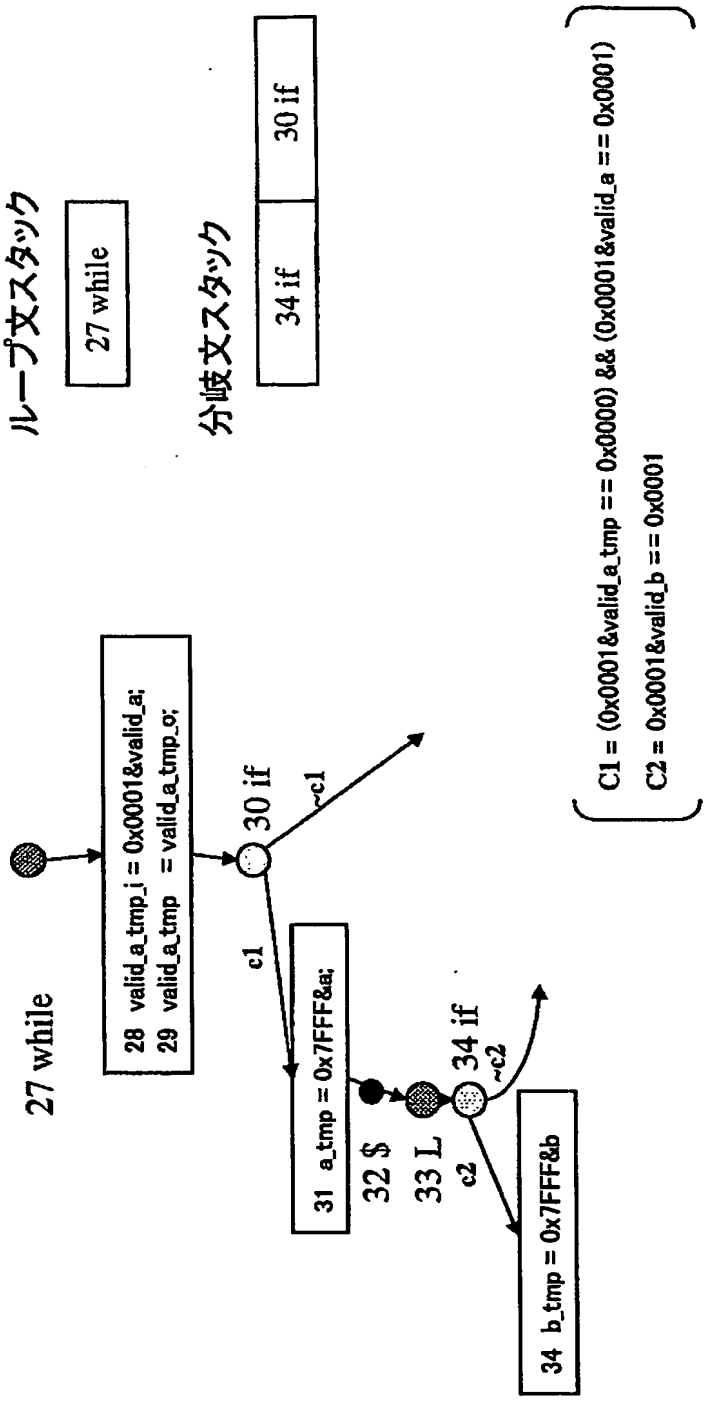
【図１２】

図１２


27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
30 if
~c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
c2
34 if
~c2
ループ文スタック
27 while
分岐文スタック
34 if
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

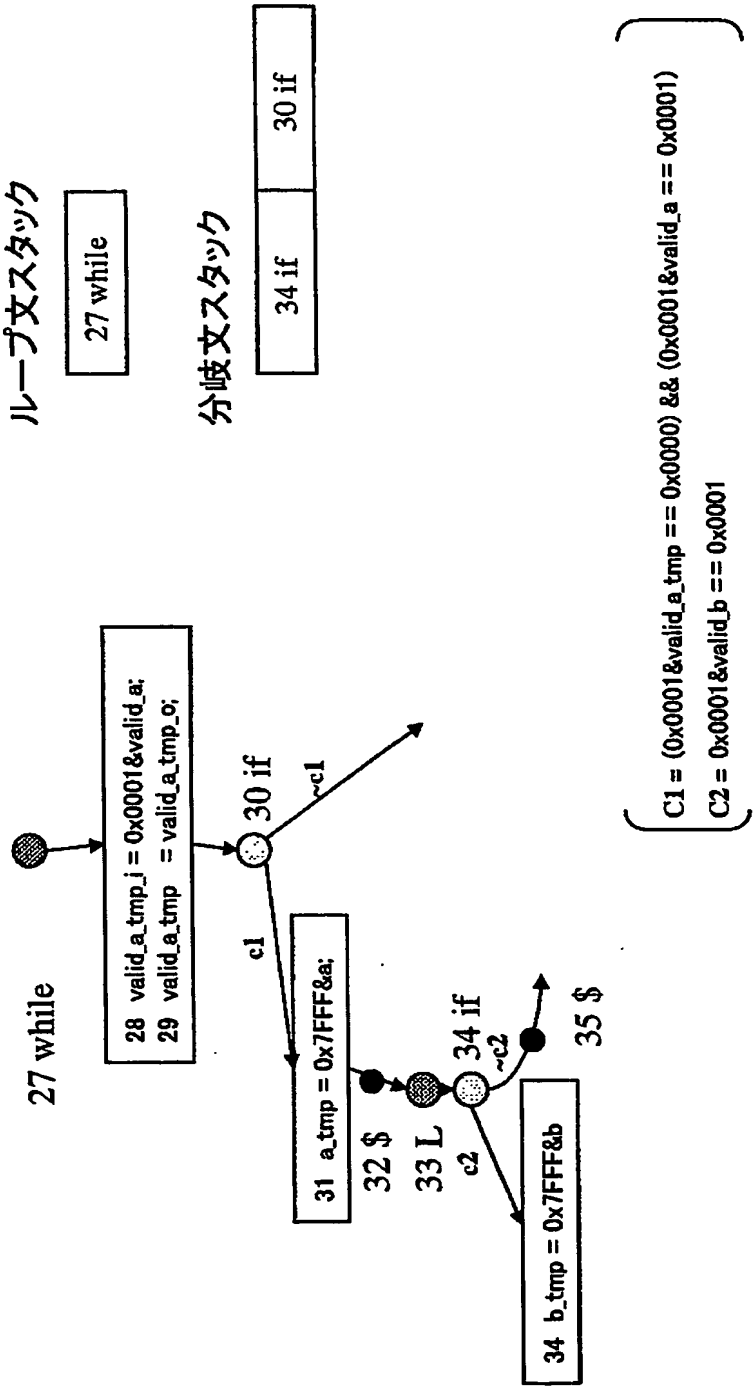
【図１３】

図１３


ループ文スタック
27 while
分岐文スタック
34 if
30 if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
34 b_tmp = 0x7FFF&b

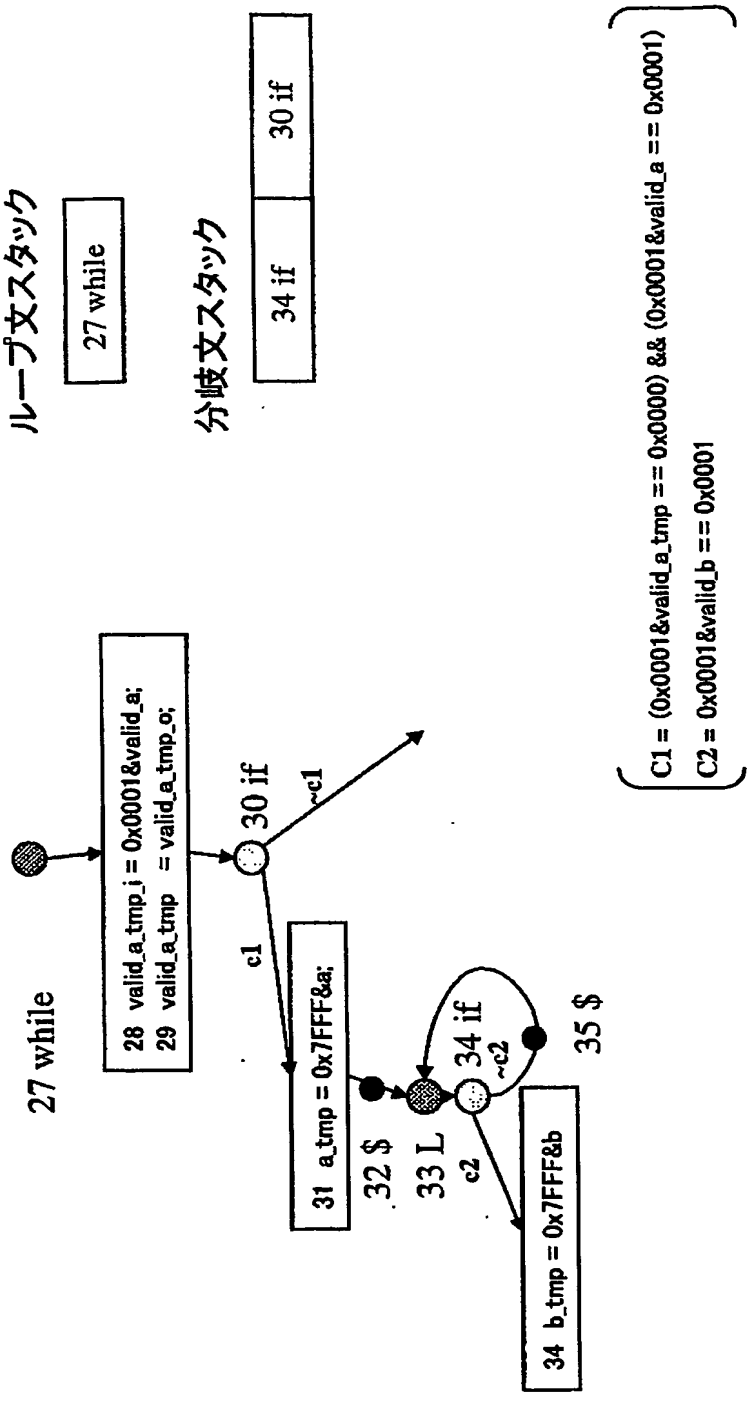
【図１４】

図１４


ループ文スタック
27 while
分岐文スタック
34 if
30 if
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

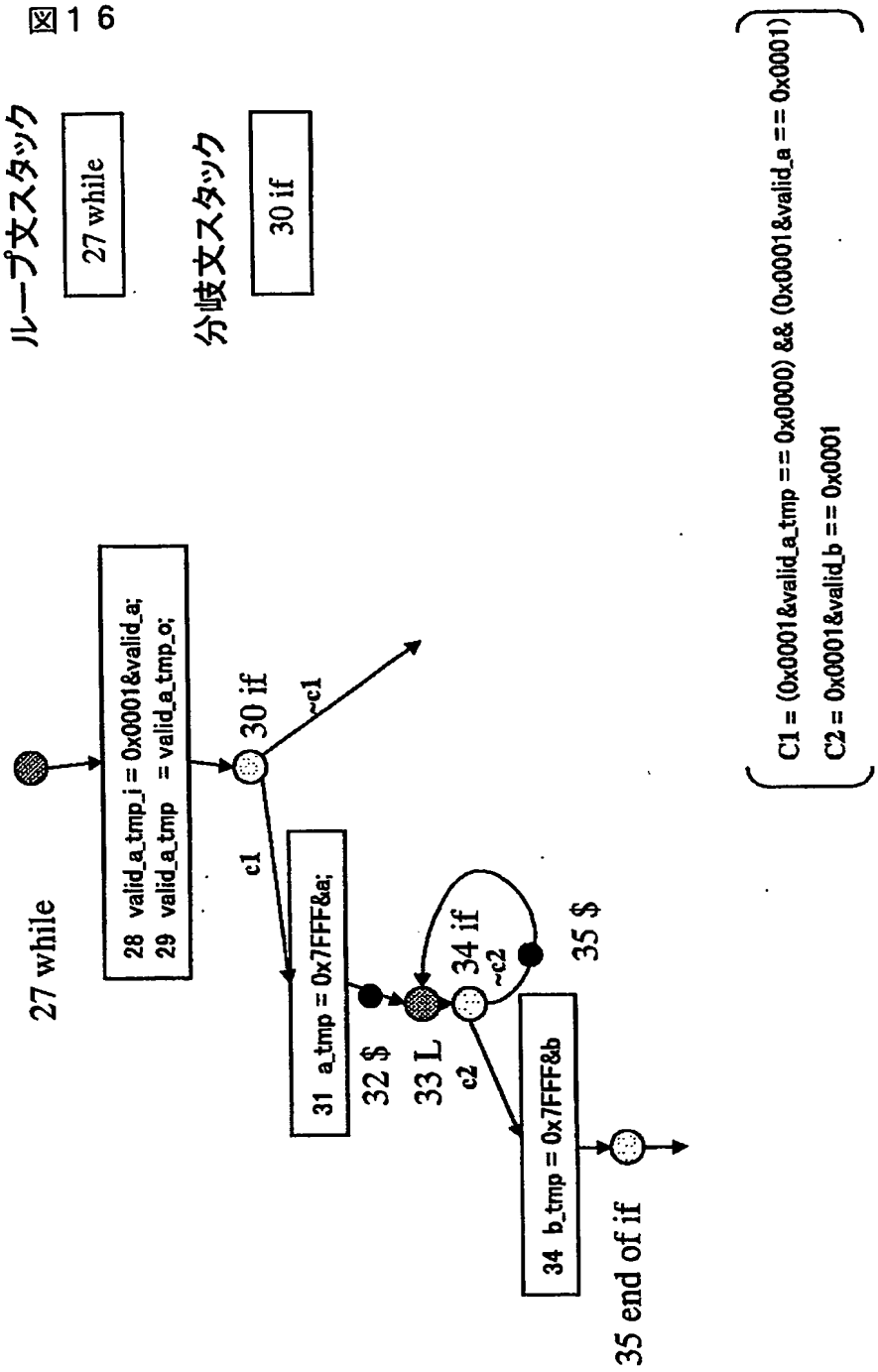
【図１５】

図１５


ループ文スタック
27 while
分岐文スタック
34 if
30 if
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

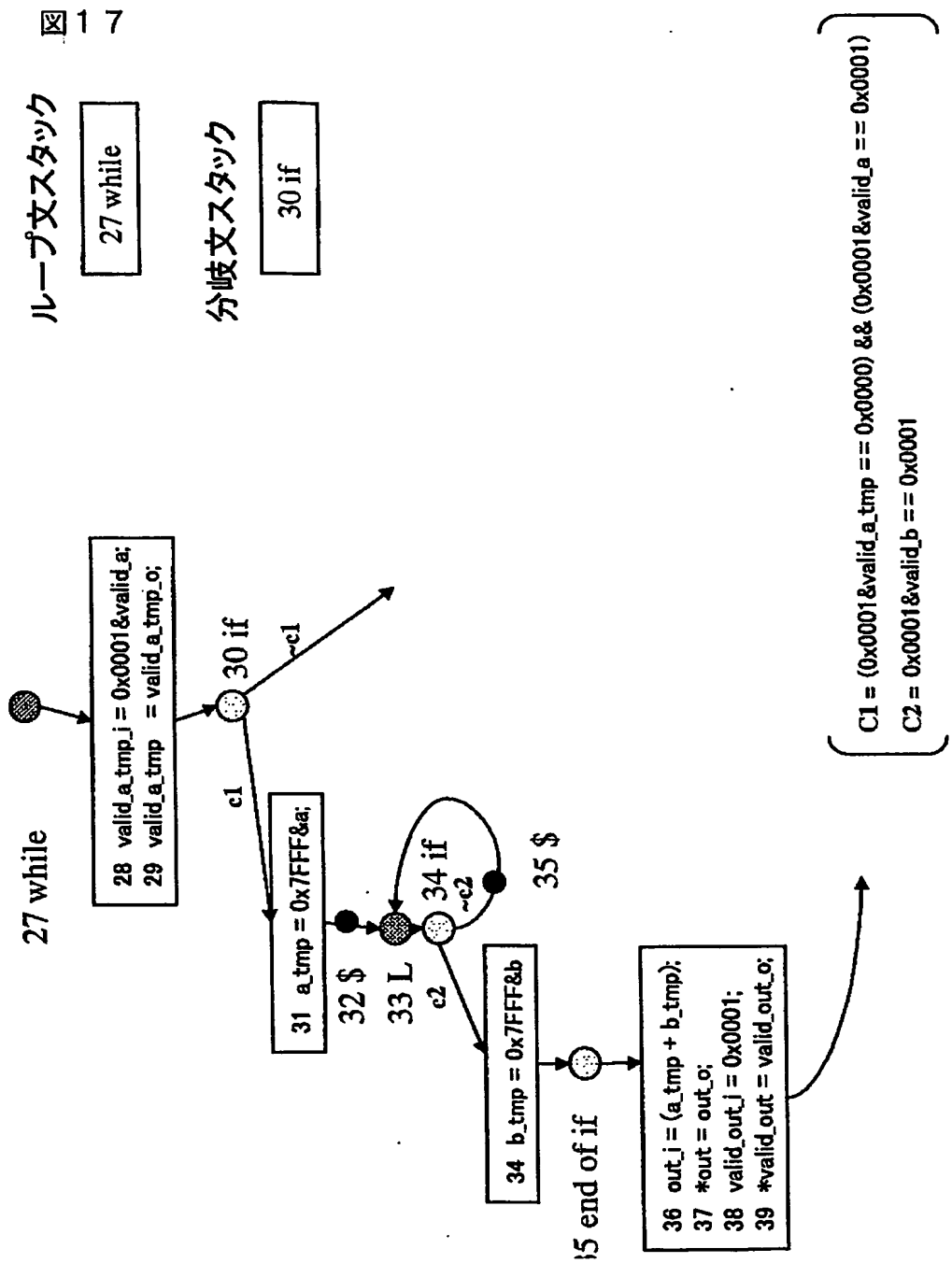
【図１６】

図１６


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
~c1
c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

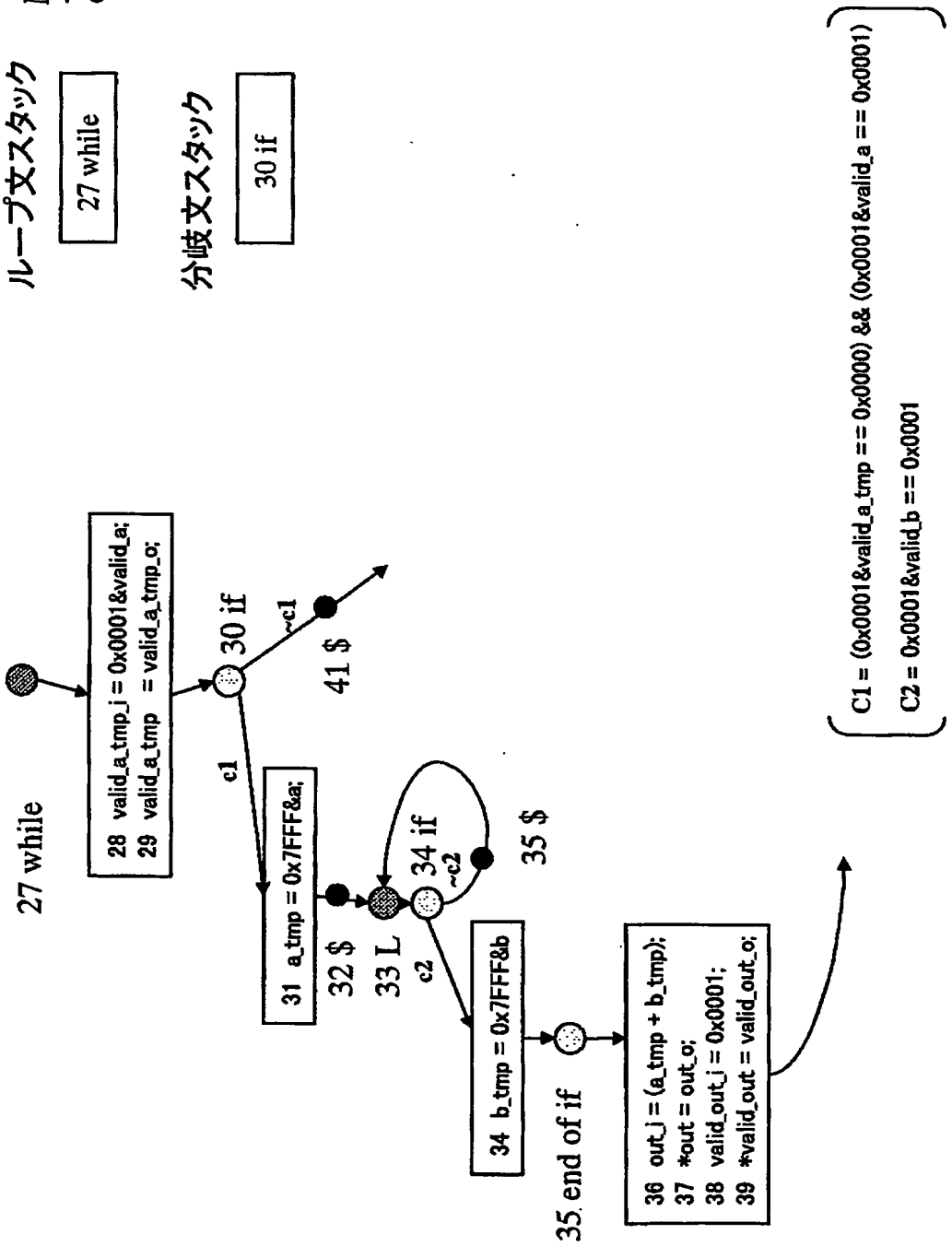
【図１７】

図１７


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

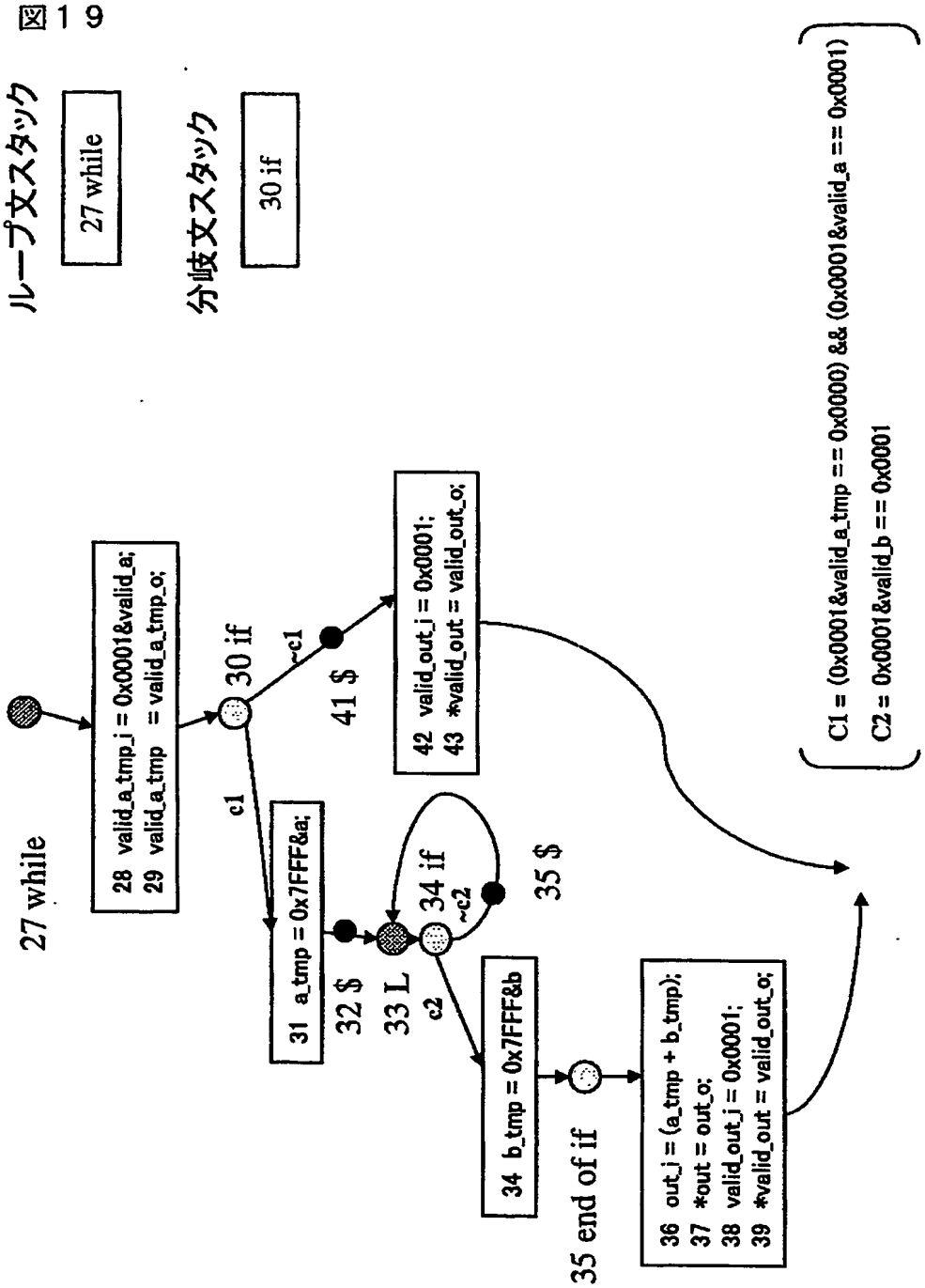
【図１８】

図１８


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_j = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

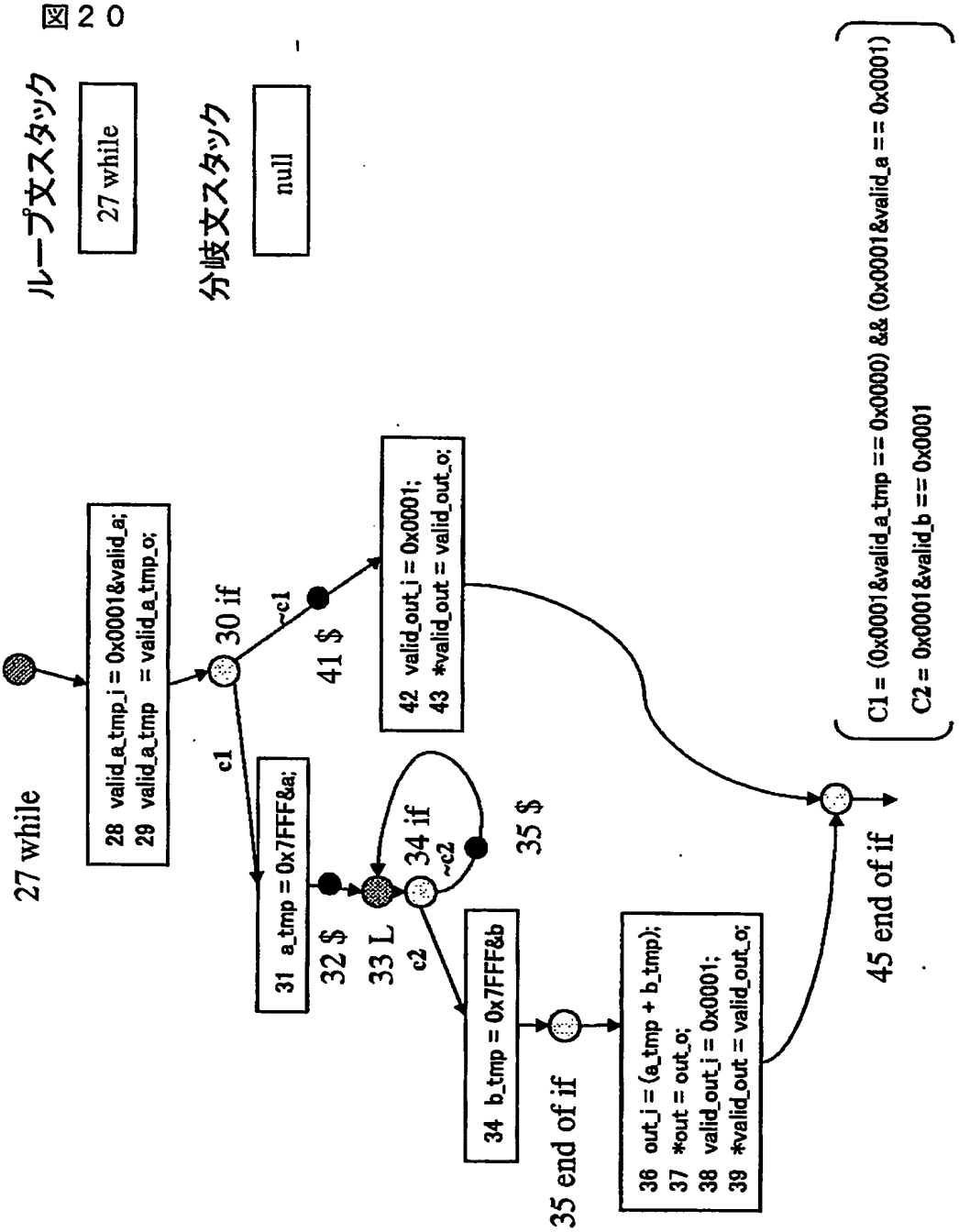
【図１９】

図１９


ループ文スタック
27 while
分岐文スタック
30 if
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

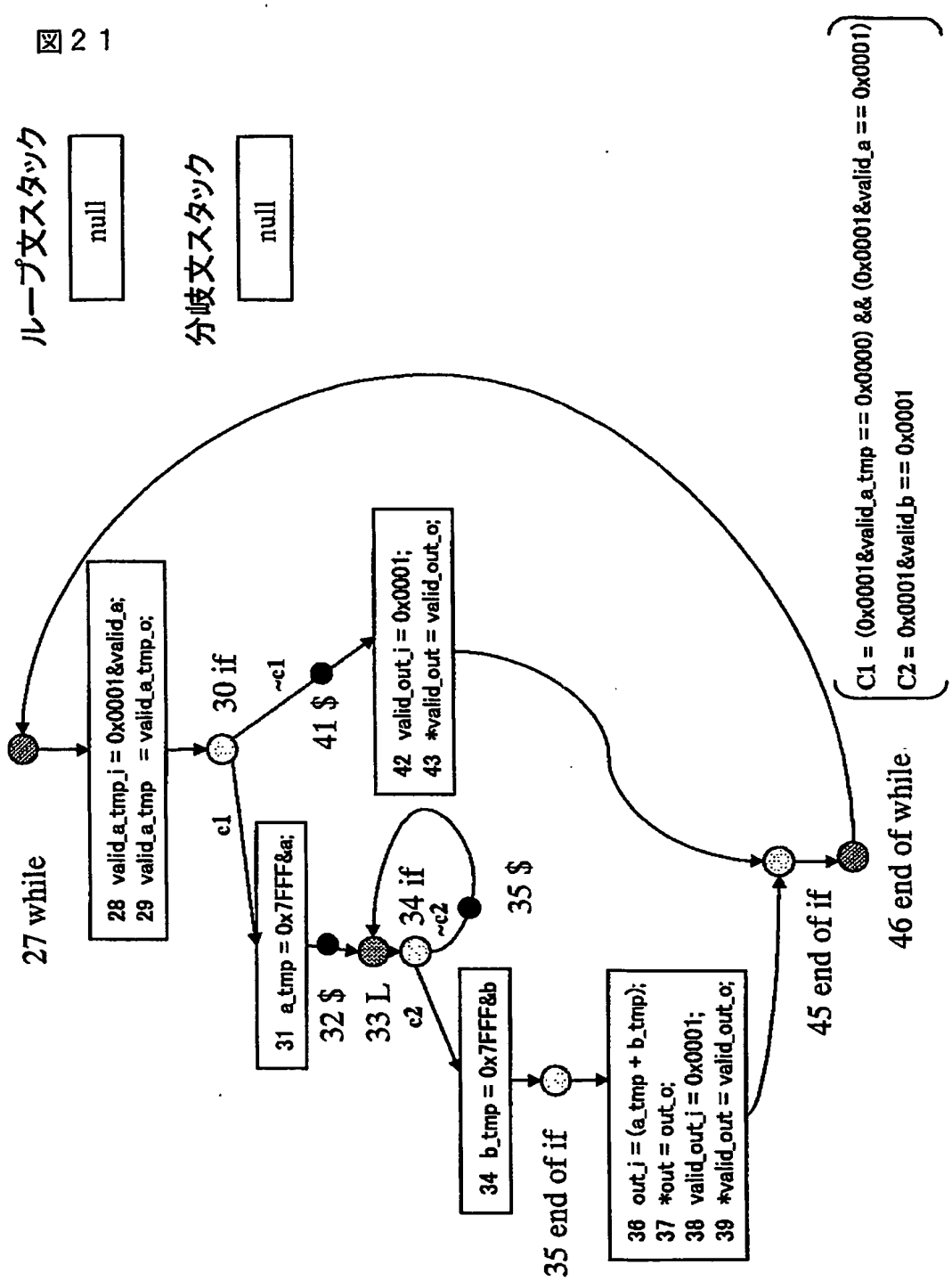
【図２０】

図２０


ループ文スタック
27 while
分岐文スタック
null
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
35 $
c2
34 b_tmp = 0x7FFF&b
35 end of if
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
45 end of if
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

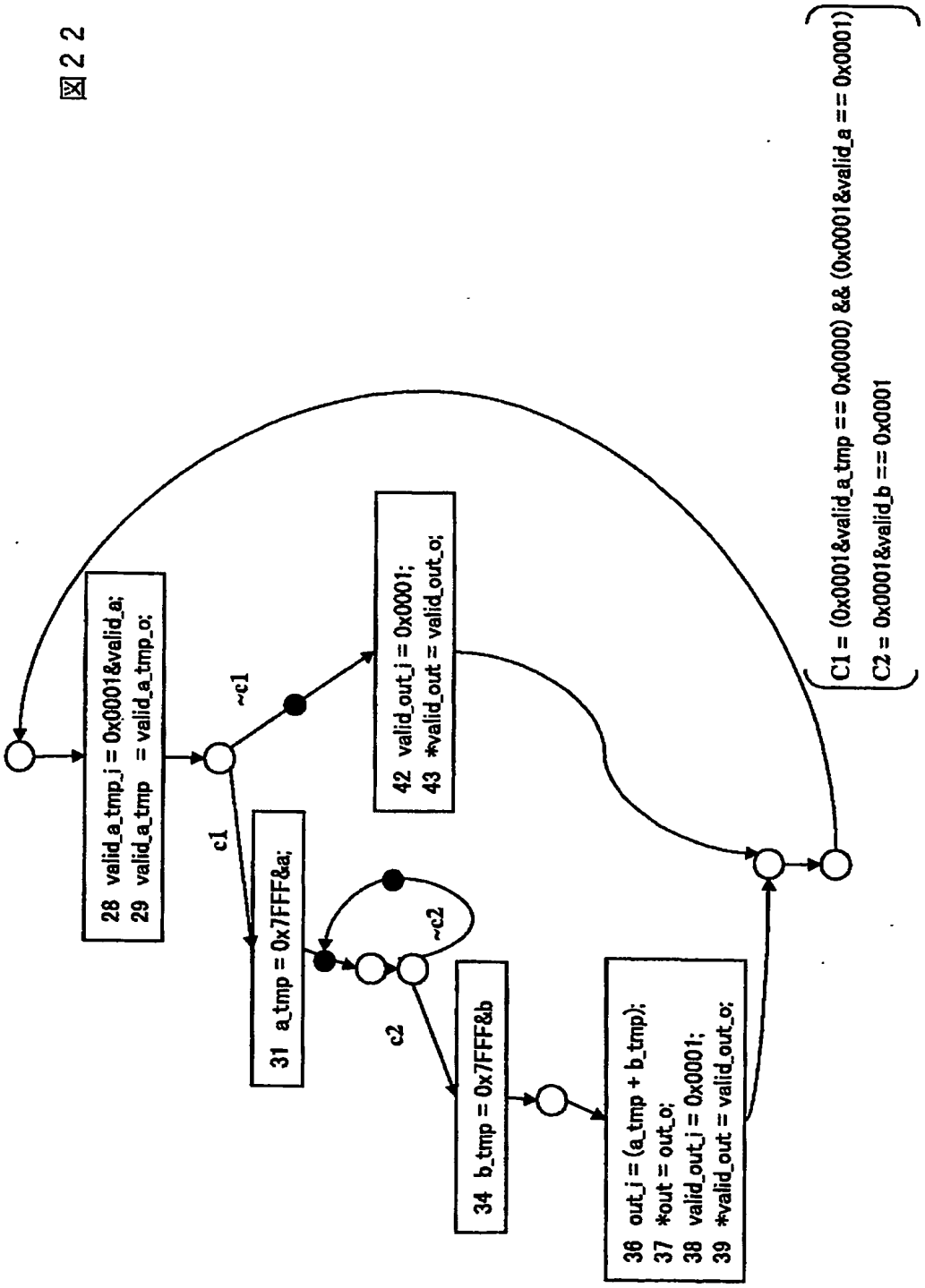
【図２１】

図２１


ループ文スタック
null
分岐文スタック
null
27 while
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
30 if
c1
~c1
41 $
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
32 $
33 L
34 if
~c2
c2
35 $
34 b_tmp = 0x7FFF&b
35 end of if
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
45 end of if
46 end of while
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

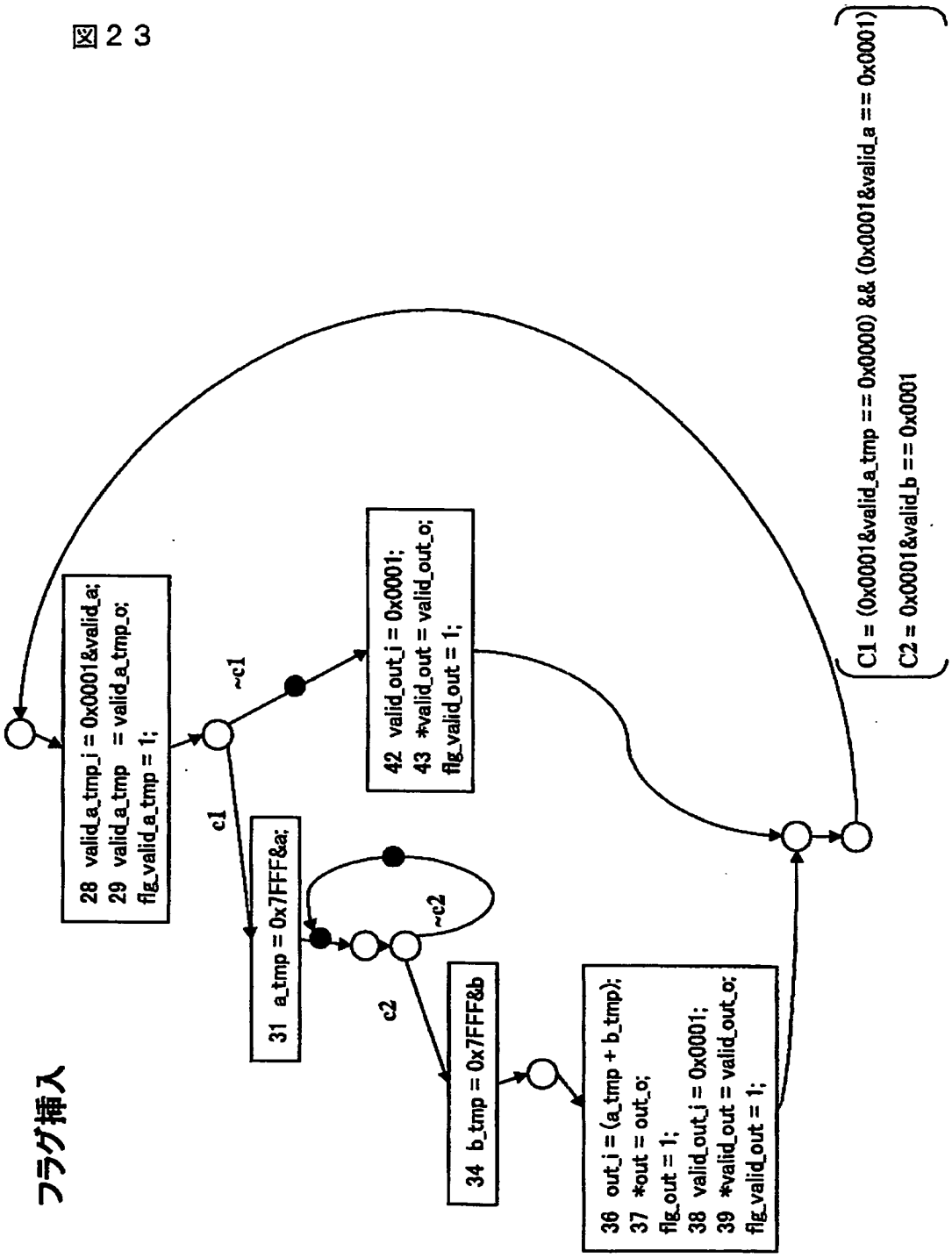
【図２２】

図２２


28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
~c1
c1
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

【図２３】

図２３


フラグ挿入
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
flg_valid_a_tmp = 1;
~c1
c1
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
flg_valid_out = 1;
31 a_tmp = 0x7FFF&a;
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
flg_out = 1;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
flg_valid_out = 1;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

【図２４】

図２４

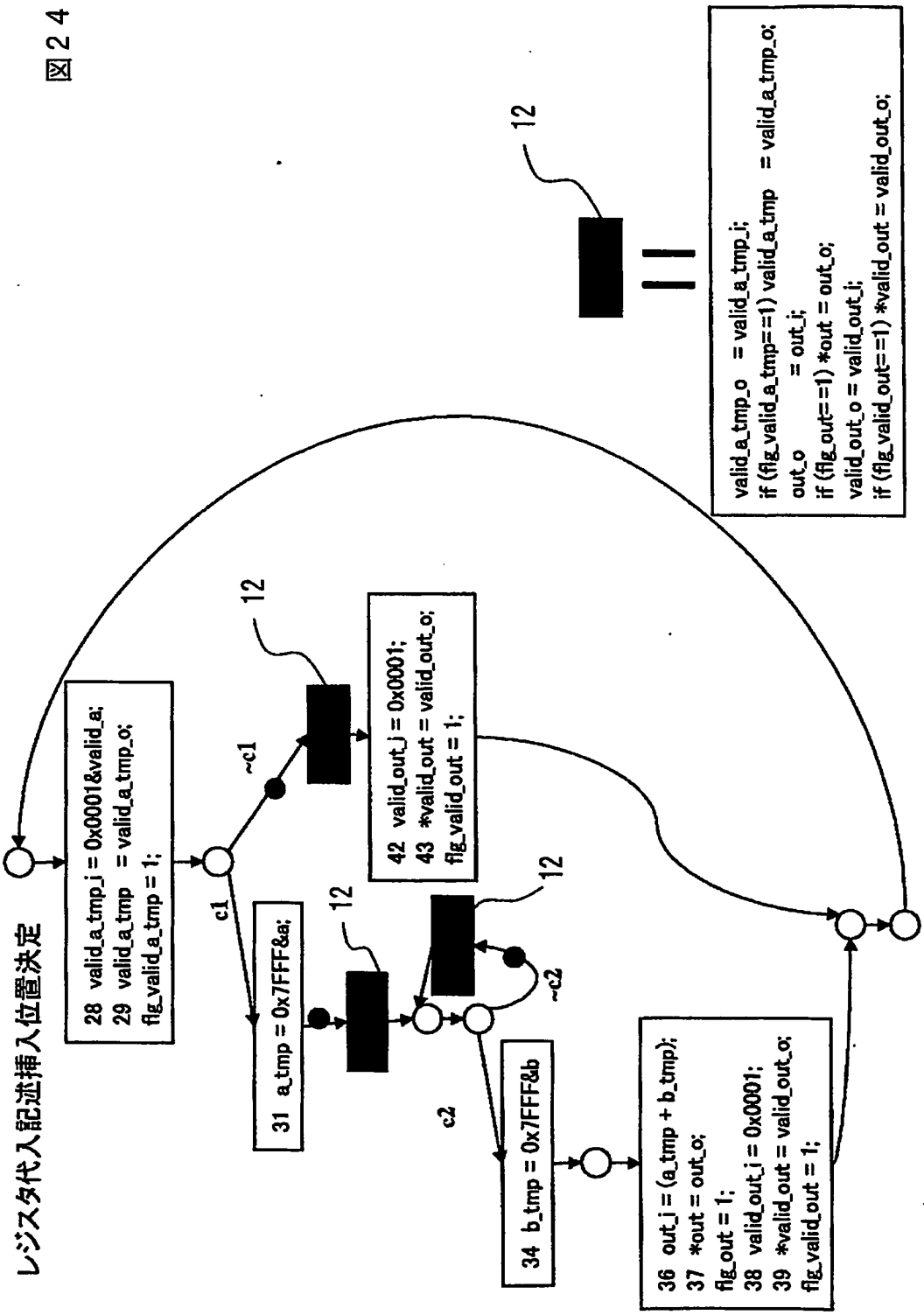

レジスタ代入記述挿入位置決定
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
flg_valid_a_tmp = 1;
c1
~c1
12
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
flg_valid_out = 1;
31 a_tmp = 0x7FFF&a;
12
c2
~c2
12
34 b_tmp = 0x7FFF&b
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
flg_out = 1;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
flg_valid_out = 1;
12
valid_a_tmp_o = valid_a_tmp_i;
if (flg_valid_a_tmp==1) valid_a_tmp = valid_a_tmp_o;
out_o = out_i;
if (flg_out==1) *out = out_o;
valid_out_o = valid_out_i;
if (flg_valid_out==1) *valid_out = valid_out_o;

図２５

```
1  #include <stdio.h>
2  void pipeline(unsigned short valid_a,
3                unsigned short valid_b,
4                unsigned short a,
5                unsigned short b,
6                unsigned short *out,
7                unsigned short *valid_out);
8  main() {
9    unsigned short valid_a, valid_b,
10                  a, b, *out, *valid_out;
11   *out       = 0x0000;
12   *valid_out = 0x0000;
13   pipeline(valid_a, valid_b, a, b, out, valid_out);
14 }
15 void pipeline(unsigned short valid_a, unsigned short valid_b,
16               unsigned short a, unsigned short b,
17               unsigned short *out, unsigned short *valid_out) {
18   unsigned short valid_a_tmp      = 0x0000;
19   unsigned short a_tmp            = 0x0000;
20   unsigned short b_tmp            = 0x0000;
     /* Added variables */
21   unsigned short valid_a_tmp_i;
22   unsigned short valid_a_tmp_o    = 0x0000;
23   unsigned short valid_out_i;
24   unsigned short valid_out_o  = 0x0000;
25   unsigned short out_i;
26   unsigned short out_o            = 0x0000;
27   unsigned short flg_valid_a_tmp = 0x0000;
28   unsigned short flg_valid_out = 0x0000;
29   unsigned short flg_out          = 0x0000;
```

【図２６】

図２６

```
30  while (1) {
     /* valid_a_tmp = $ valid_a; */
31   valid_a_tmp_i = 0x0001&valid_a;        /* Refined */
32   valid_a_tmp   = valid_a_tmp_o;         /* Refined */
33   flg_valid_a_tmp = 1;
34   if ((0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)) {
35     a_tmp = 0x7FFF&a;
       /* $ */
       /* BEGIN : Register Assignment */
36     valid_a_tmp_o   = valid_a_tmp_i;
37     if (flg_value_a_tmp == 1) valid_a_tmp     = valid_a_tmp_o;
38     out_o           = out_i;
39     if (flg_out==1) *out = out_o;
40     valid_out_o = valid_out_i;
41     if (flg_valid_out==1) *valid_out = valid_out_o;
       /* END   : Register Assignment */
42   L :
```

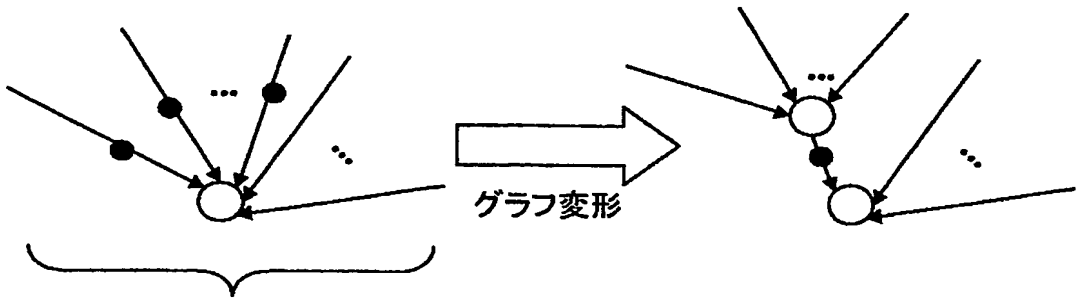
【図２７】

図２７

```
43    if (0x0001&valid_b == 0x0001) b_tmp = 0x7FFF&b;
44    else {
        /* $ */
        /* BEGIN : Register Assignment */
45      valid_a_tmp_o   = valid_a_tmp_i;
46      valid_a_tmp     = valid_a_tmp_o;
47      out_o           = out_i;
48      if (flg_out==1) *out = out_o;
49      valid_out_o = valid_out_i;
50      if (flg_valid_out==1) *valid_out = valid_out_o;
        /* END   : Register Assignment */
51      goto L;
52    }
      /* *out = $ (a_tmp + b_tmp); */
53    out_i     = a_tmp + b_tmp;          /* Refined */
54    *out      = out_o;                  /* Refined */
55    flg_out   = 1;                      /* Added */
      /* *valid_out = $ 0x0001; */
56    valid_out_i    = 0x0001;            /* Refined */
57    *valid_out     = valid_out_o;       /* Refined */
58    flg_valid_out  = 1;                 /* Added */
59    } else {
        /* $ */
        /* BEGIN : Register Assignment */
60      valid_a_tmp_o   = valid_a_tmp_i;
61      valid_a_tmp     = valid_a_tmp_o;
62      out_o           = out_i;
63      if (flg_out==1) *out = out_o;
64      valid_out_o = valid_out_i;
65      if (flg_valid_out==1) *valid_out = valid_out_o;
        /* END   : Register Assignment */
        /* *valid_out = $ 0x0000; */
66      valid_out_i = 0x0000;               /* Refined */
67      *valid_out     = valid_out_o;       /* Refined */
68      flg_valid_out  = 1;                 /* Added */
69    }
70  }
71 }
```

【図２８】

図２８



ループ開始・終了ノード、条件分岐開始・終了ノード、
ラベル分岐ノードの何れかのノード

【図２９】

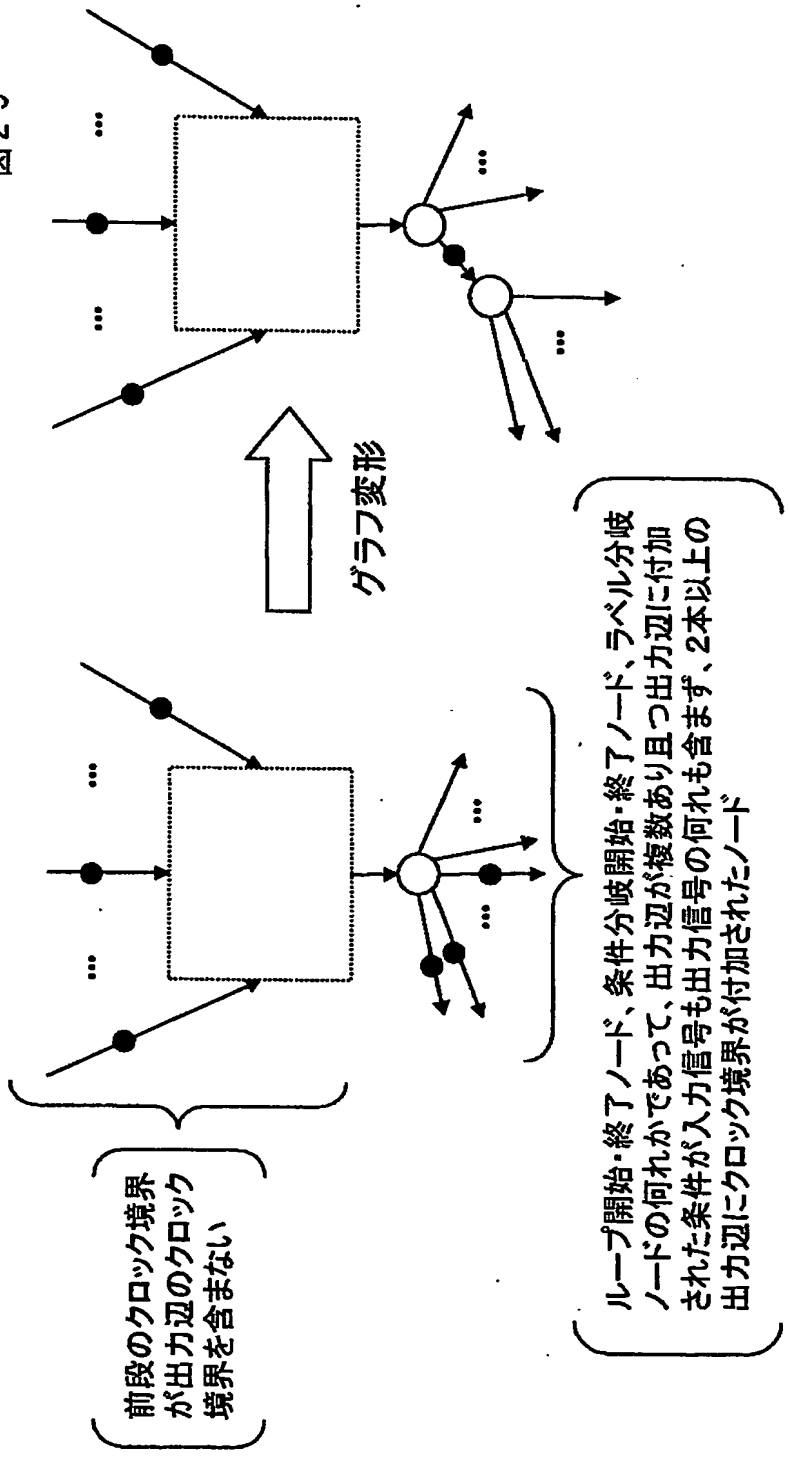
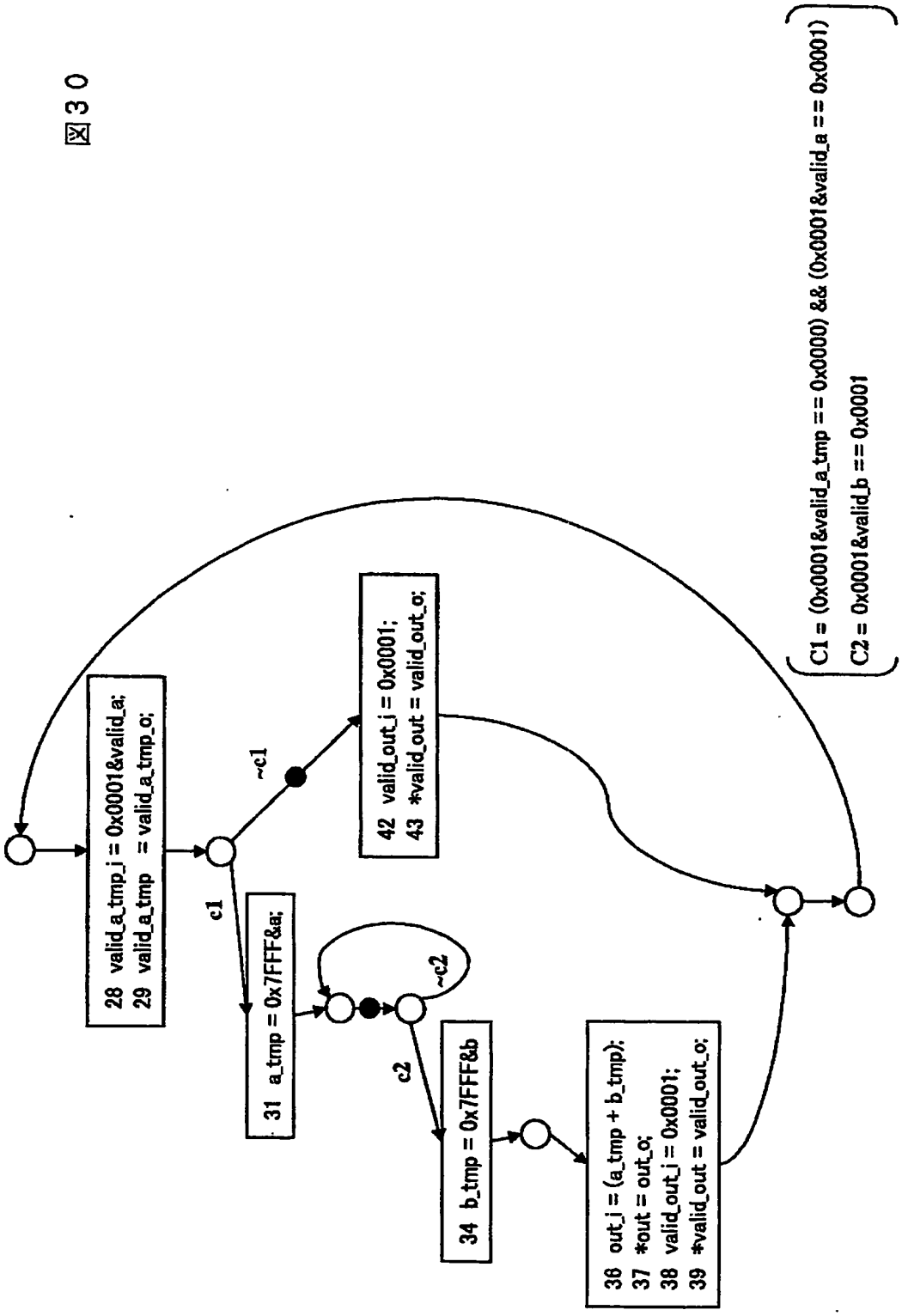


図２９
グラフ変形
前段のクロック境界が出力辺のクロック境界を含まない
ループ開始・終了ノード、条件分岐開始・終了ノード、ラベル分岐ノードの何れかであって、出力辺が複数あり且つ出力辺に付加された条件が入力信号も出力信号の何れも含まず、2本以上の出力辺にクロック境界が付加されたノード

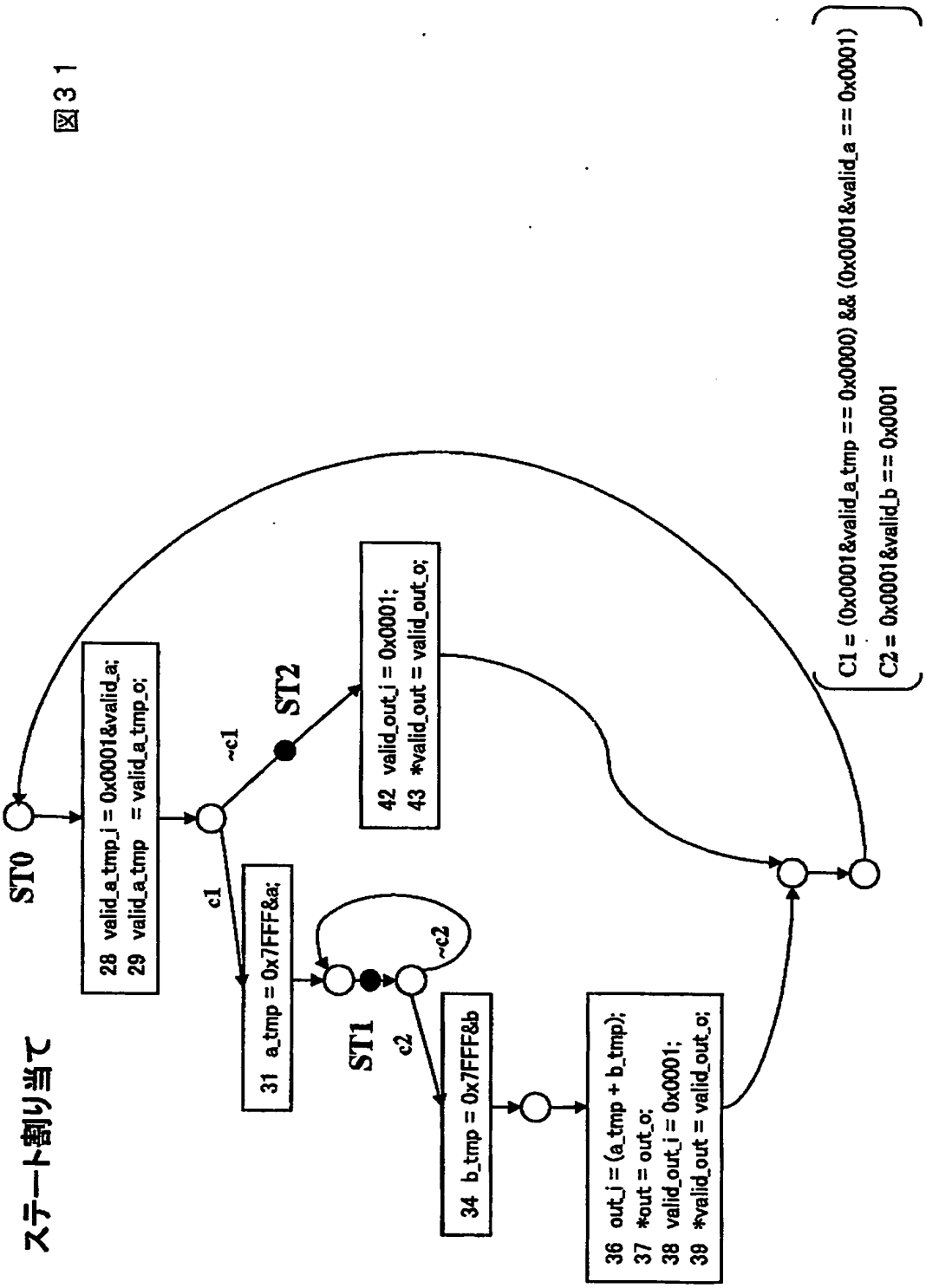
【図３０】

図３０


C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
~c1
c1
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
31 a_tmp = 0x7FFF&a;
~c2
c2
34 b_tmp = 0x7FFF&b
38 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;

【図３１】

図３１


ステート割り当て
ST0
ST1
ST2
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
~c1
c1
31 a_tmp = 0x7FFF&a;
~c2
c2
34 b_tmp = 0x7FFF&b
36 out_i = (a_tmp + b_tmp);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

【図３２】

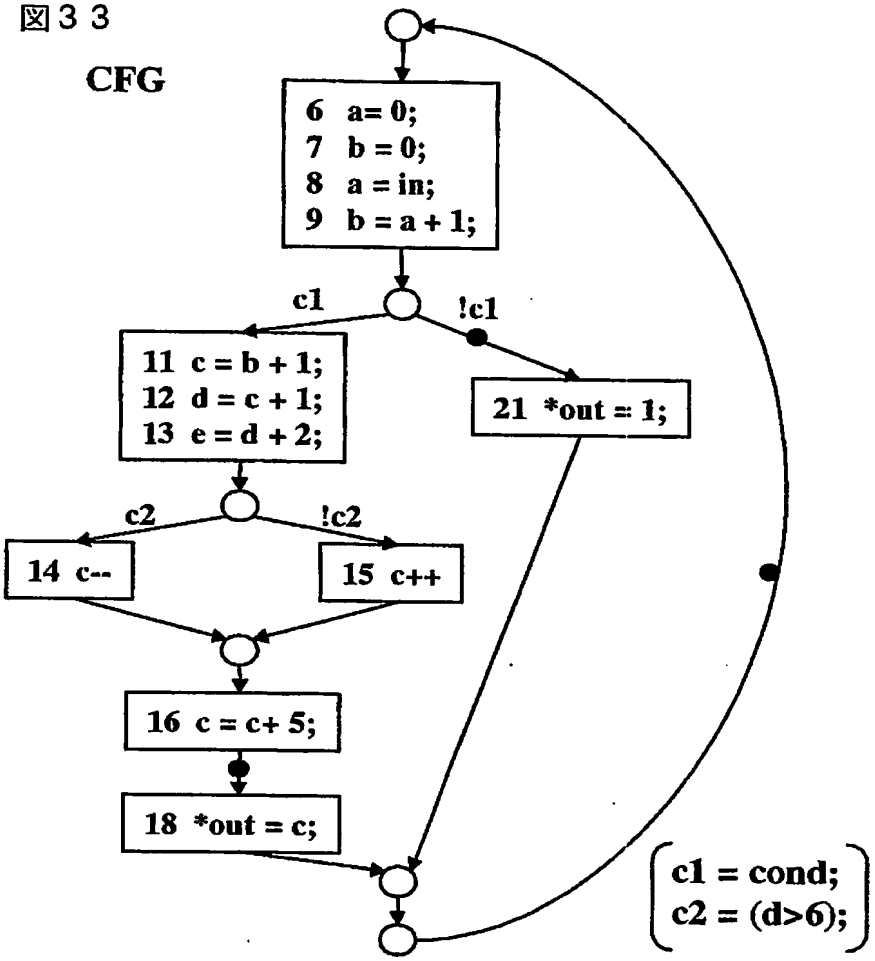
図３２　　コード最適化（別の例）

```
1  void foo(unsigned short in,
2           unsigned short cond,
3           unsigned short *out) {
4   unsigned short a, b, c, d, e;
5   while(1) {
6    a= 0;
7    b = 0;
8    a = in;
9    b = a + 1;
10    if (cond) {
11     c = b + 1;
12     d = c + 1;
13     e  = d + 2;
14     if (d > 6) c--;
15     else       c++;
16     c = c + 5;
17     $
18     *out = c;
19    } else {
20     $
21     *out = 1;
22    }
23    $
24 }
```

【図３３】

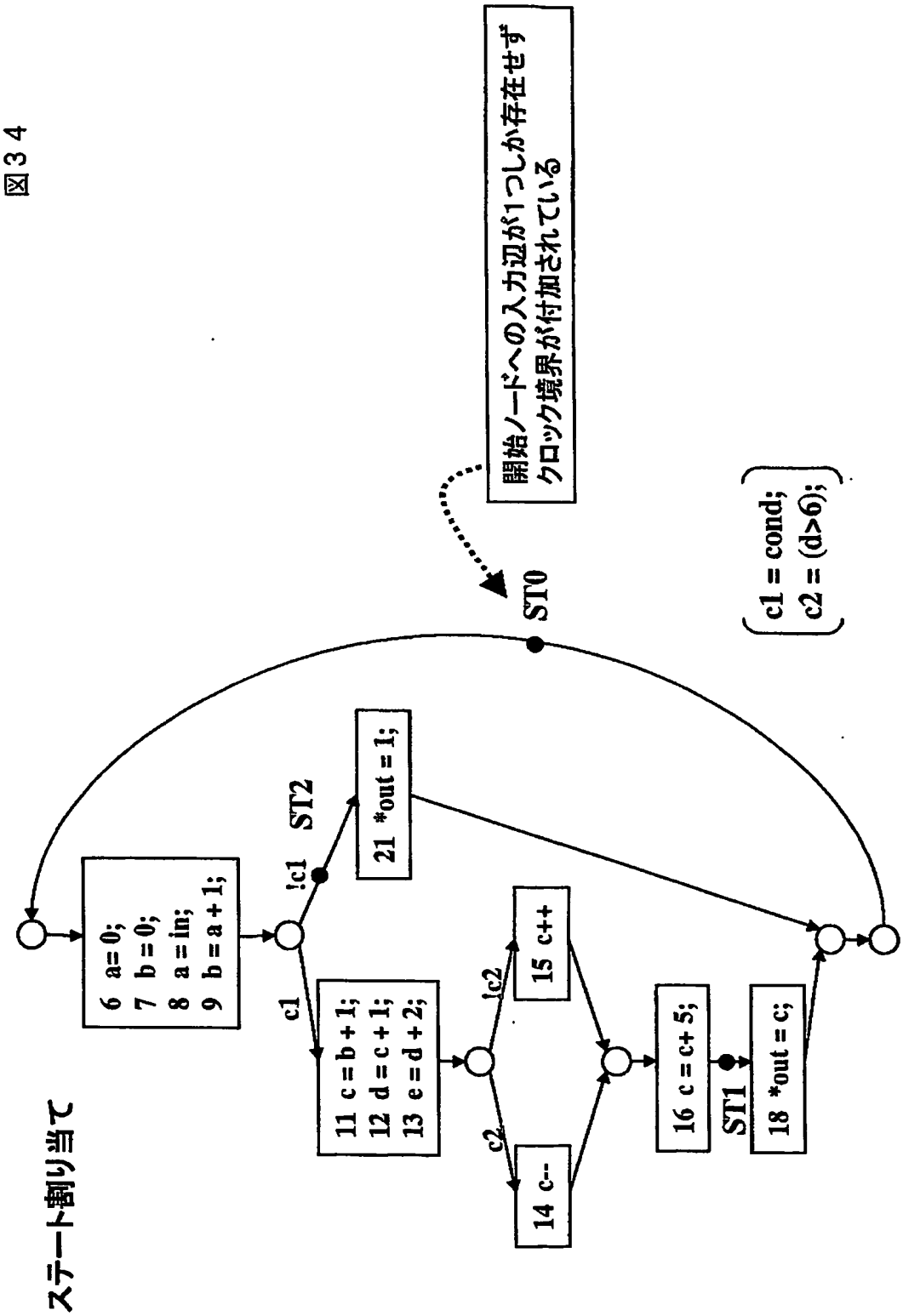
図３３

CFG


6 a= 0;
7 b = 0;
8 a = in;
9 b = a + 1;
c1
!c1
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
21 *out = 1;
c2
!c2
14 c--
15 c++
16 c = c+ 5;
18 *out = c;
c1 = cond;
c2 = (d>6);

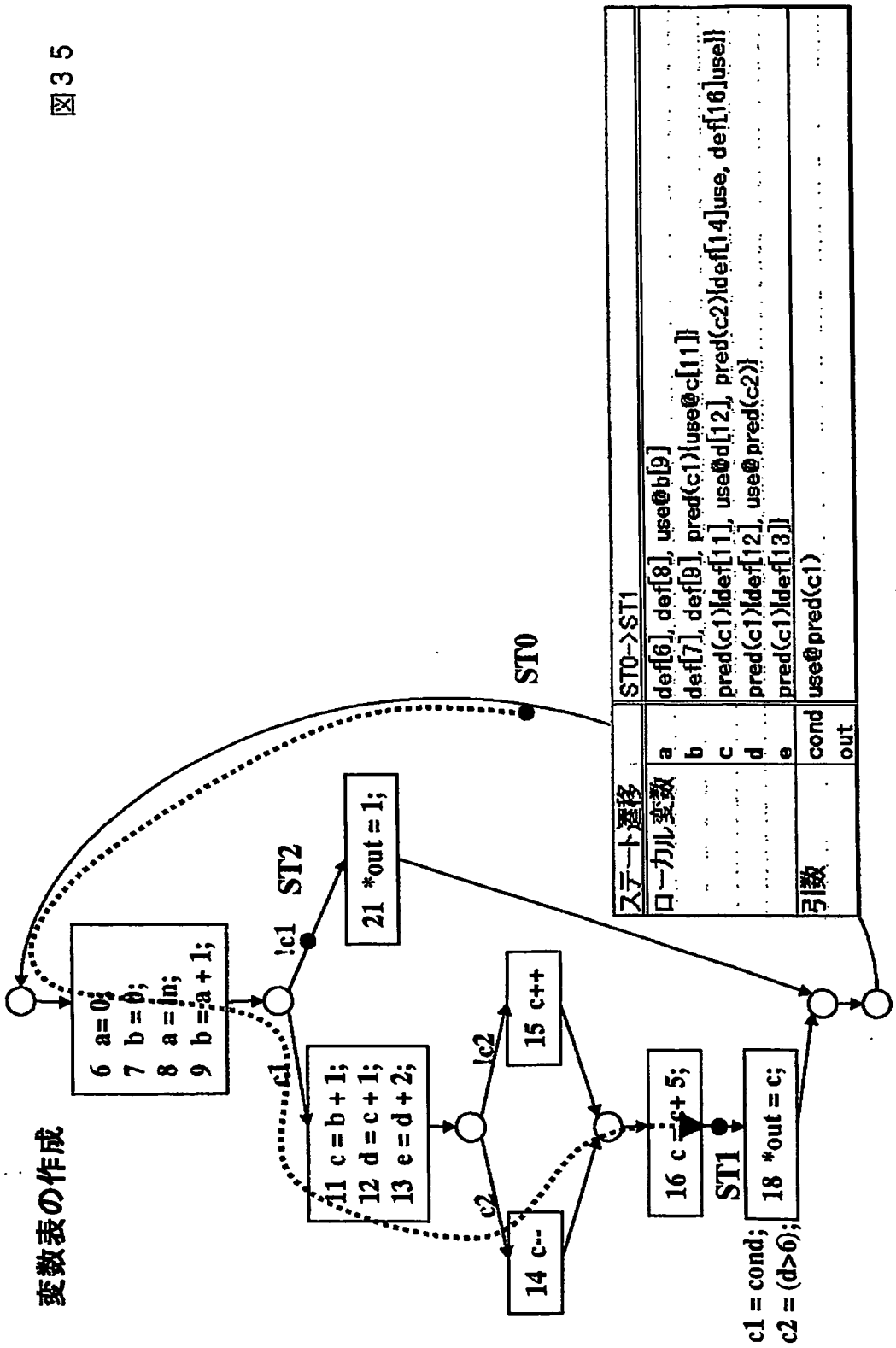
【図３４】

図３４


開始ノードへの入力辺が1つしか存在せず
クロック境界が付加されている
ST0
c1 = cond;
c2 = (d>6);
6 a=0;
7 b=0;
8 a=in;
9 b=a+1;
!c1
ST2
21 *out=1;
c1
11 c=b+1;
12 d=c+1;
13 e=d+2;
!c2
15 c++
c2
14 c--
16 c=c+5;
ST1
18 *out=c;
ステート割り当て

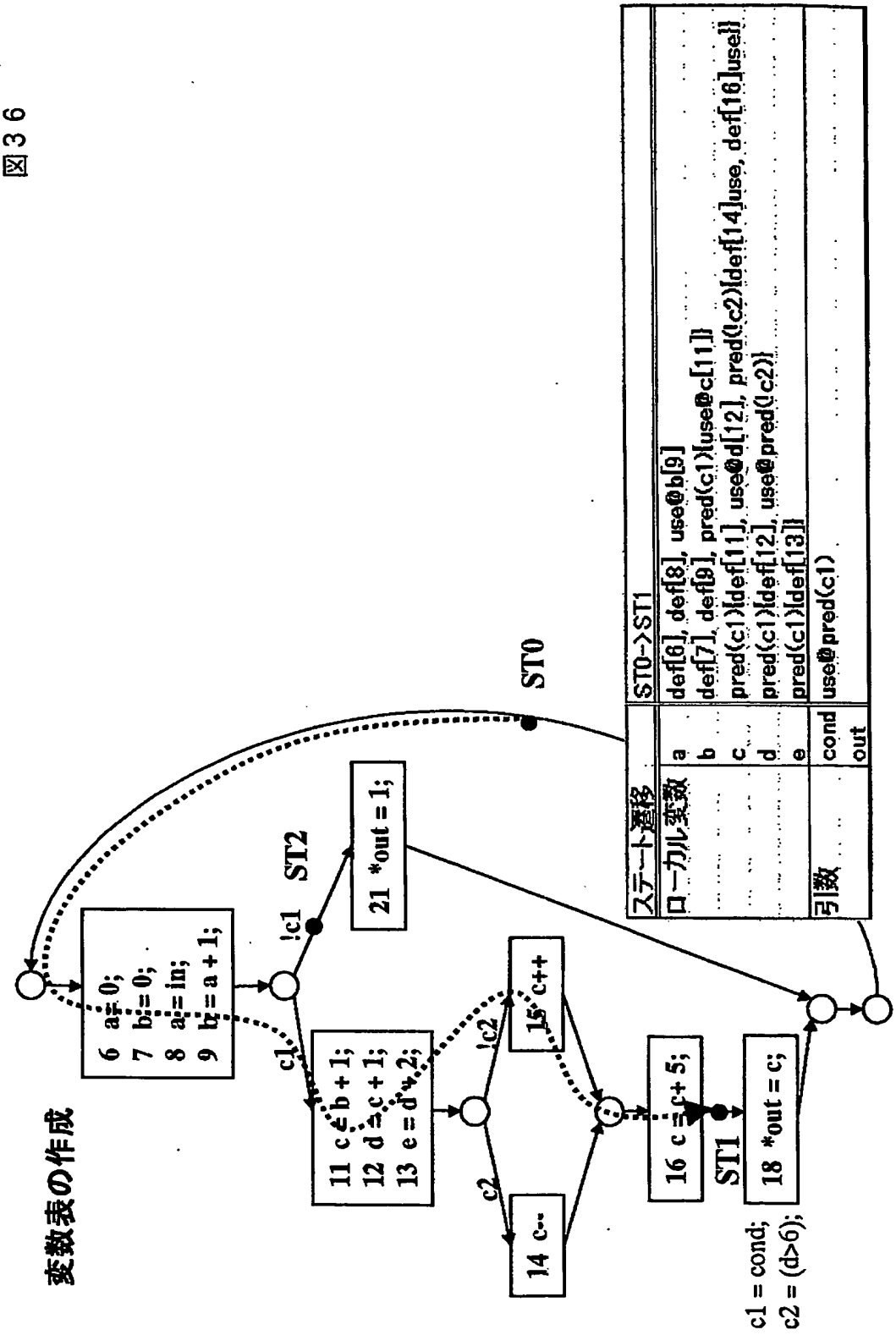
【図３５】

図３５


変数表の作成
6 a=0;
7 b=b;
8 a=in;
9 b=a+1;
!c1
ST2
21 *out=1;
c1
11 c=b+1;
12 d=c+1;
13 e=d+2;
!c2
15 c++
c2
14 c--
16 c=c+5;
ST1
18 *out=c;
c1 = cond;
c2 = (d>6);
ST0
ステート遷移
ST0->ST1
ローカル変数
a
def[6], def[8], use@b[9]
b
def[7], def[9], pred(c1)!use@c[11]
c
pred(c1)!def[11], use@d[12], pred(c2)!def[14]use, def[16]use
d
pred(c1)!def[12], use@pred(c2)
e
pred(c1)!def[13]
引数
cond
use@pred(c1)
out

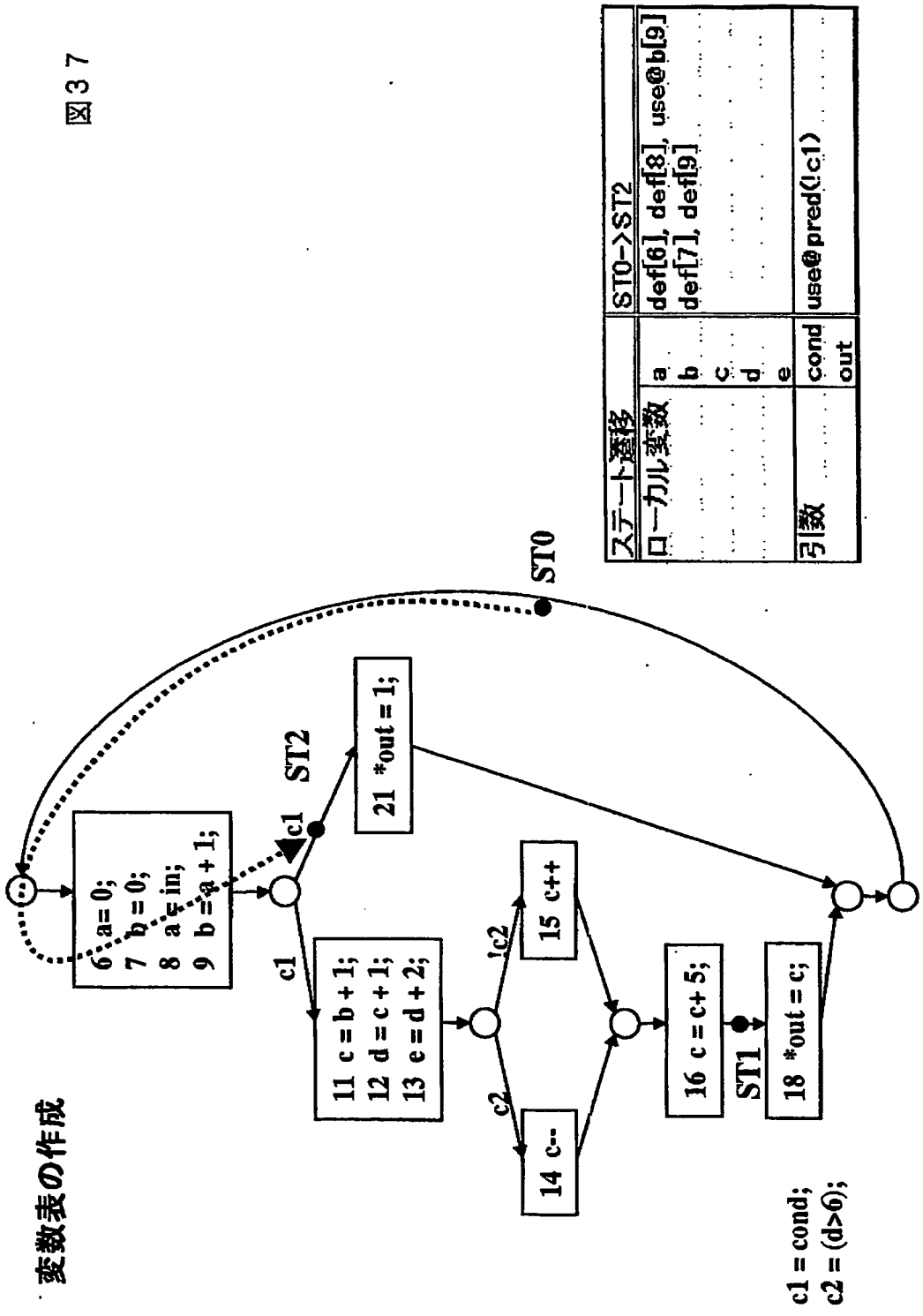
【図３６】

図３６


変数表の作成
6 a = 0;
7 b = 0;
8 a = in;
9 b = a + 1;
c1
!c1
ST2
11 c = b + 1;
12 d = c + 1;
13 e = d + 2;
21 *out = 1;
c2
!c2
14 c--
15 c++
16 c = c + 5;
ST1
18 *out = c;
ST0
c1 = cond;
c2 = (d>6);
ステート遷移
ST0->ST1
ローカル変数
a
def[6], def[8], use@b[9]
b
def[7], def[9], pred(c1)/use@c[11]
c
pred(c1)/def[11], use@d[12], pred(!c2)/def[14]use, def[16]use
d
pred(c1)/def[12], use@pred(!c2)
e
pred(c1)/def[13]
引数
cond
use@pred(c1)
out

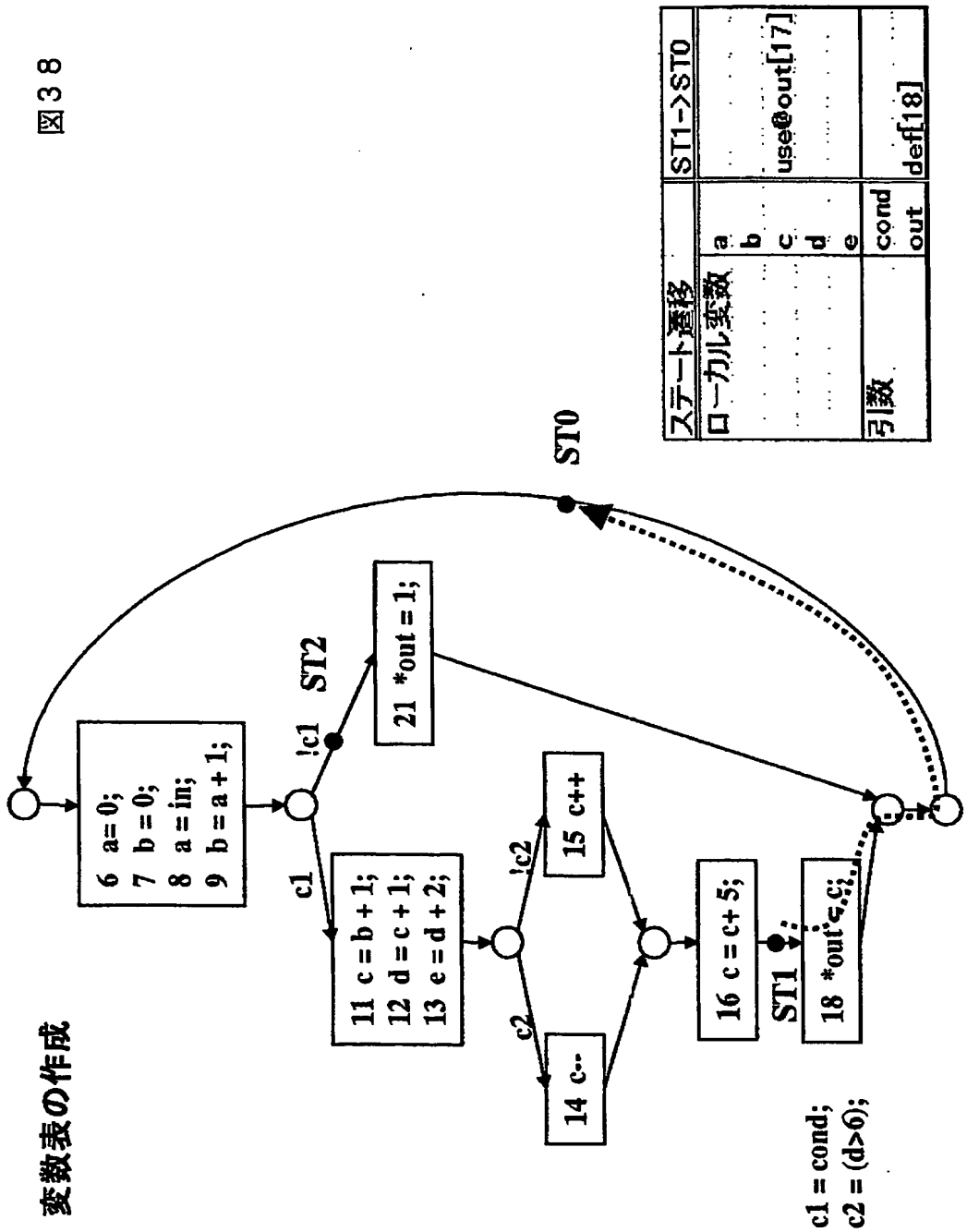
【図３７】

図３７


変数表の作成
6 a=0;
7 b=0;
8 a≦in;
9 b=a+1;
c1
ST2
21 *out=1;
11 c=b+1;
12 d=c+1;
13 e=d+2;
c2
15 c++
14 c--
16 c=c+5;
ST1
18 *out=c;
ST0
c1=cond;
c2=(d>6);
ステート遷移
ST0->ST2
ローカル変数
a
b
c
d
e
def[6], def[8], use@b[9]
def[7], def[9]
引数
cond
out
use@pred(!c1)

【図３８】

図３８


ステート遷移
ST1->ST0
ローカル変数
a
b
c
d
e
use@out[17]
引数
cond
out
def[18]
ST0
ST2
!c1
c1
6 a=0;
7 b=0;
8 a=in;
9 b=a+1;
21 *out=1;
11 c=b+1;
12 d=c+1;
13 e=d+2;
!c2
c2
15 c++
14 c--
16 c=c+5;
ST1
18 *out=c;;
変数表の作成
c1 = cond;
c2 = (d>6);

変数表の作成

図４０

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[6], def[8], use@b[9]	def[6], def[8], use@b[9]		
	b	def[7], def[9], pred(c1){use@c[11]}	def[7], def[9]		
	c	pred(c1){def[11], use@d[12], pred(c2){def[14]use, def[16]use}}		use@out[17]	
	d	pred(c1){def[12], use@pred(c2)}			
	e	pred(c1){def[13]}			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]

ステート遷移		ST0->ST1
ローカル変数	a	def[6], def[8], use@b[9]
	b	def[7], def[9], pred(c1){use@c[11]}
	c	pred(c1){def[11], use@d[12], pred(!c2){def[14]use, def[16]use}}
	d	pred(c1){def[12], use@pred(!c2)}
	e	pred(c1){def[13]}
引数	cond	use@pred(c1)
	out	

def[n]　　：n行目で変数定義されている事を表す。
use@var[m]　：m行目で変数varへの代入に用いられている事を表す。
pred(cond){...}　：条件condの分岐が成立した場合、{...}が実施される事を表す。
def[l]use　　：l行目で自変数への代入に用いられている事を表す。
use@pred(cond)：条件condで用いられている事を表す。

【図４１】

図４１

冗長ステートメント削除

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[6], def[8], use@b[9]	def[6], def[8], use@b[9]		
	b	def[7], def[9], pred(c1)|use@c[11]|	def[7], def[9]		
	c	pred(c1)|def[11], use@d[12], pred(c2)|def[14]use, def[16]use||		use@out[17]	
	d	pred(c1)|def[12], use@pred(c2)|			
	e	pred(c1)|def[13]|			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]

ステート遷移		ST0->ST1
ローカル変数	a	def[6], def[8], use@b[9]
	b	def[7], def[9], pred(c1)|use@c[11]|
	c	pred(c1)|def[11], use@d[12], pred(c2)|def[14]use, def[16]use||
	d	pred(c1)|def[12], use@pred(c2)|
	e	pred(c1)|def[13]|
引数	cond	use@pred(c1)
	out	

【図４２】

図４２

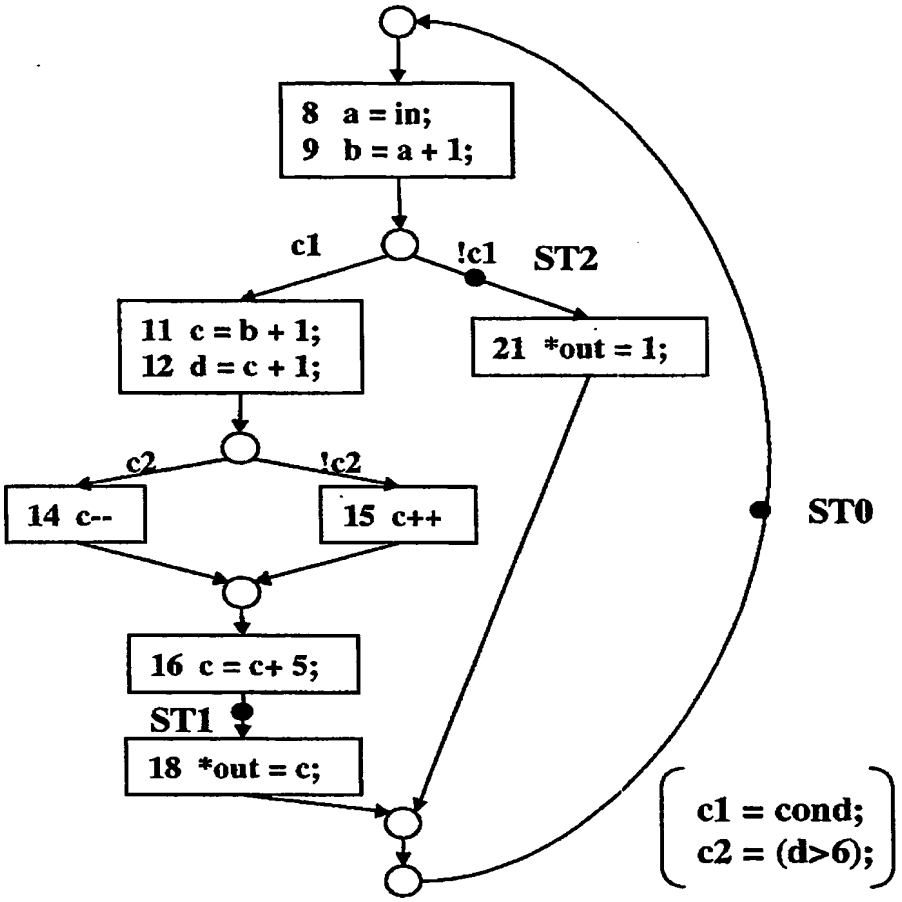
冗長ステートメント削除

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[8], use@b[9]	def[8], use@b[9]		
	b	def[9], pred(c1)|use@c[11]|	def[9]		
	c	pred(c1)|def[11], use@d[12], pred(c2)|def[14]|use, def[16]|use||		use@out[17]	
	d	pred(c1)|def[12], use@pred(c2)|			
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]
ステート遷移		ST0->ST1			
ローカル変数	a	def[8], use@b[9]			
	b	def[9], pred(c1)|use@c[11]|			
	c	pred(c1)|def[11], use@d[12], pred(!c2)|def[14]|use, def[16]|use||			
	d	pred(c1)|def[12], use@pred(!c2)|			
引数	cond	use@pred(c1)			
	out				

【図４３】

図４３

冗長ステートメント削除



【図４４】

図４４

ローカル変数削除

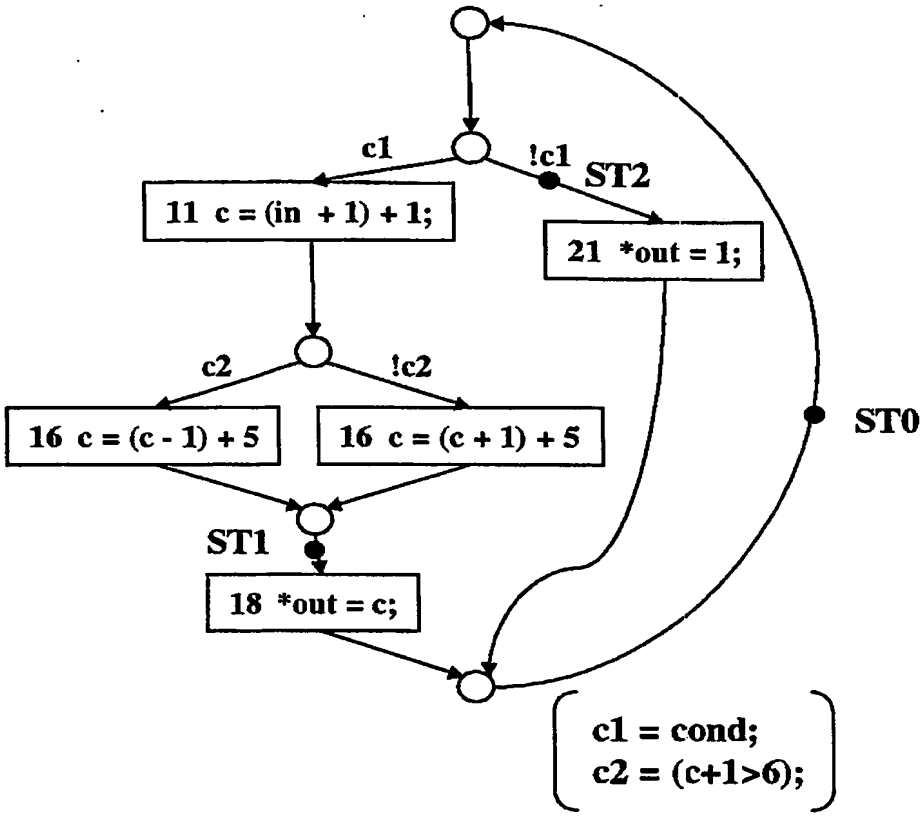
ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	a	def[8], use@b[9]	def[8], use@b[9]		
	b	def[9], pred(c1)|use@c[11]|	def[9]	use@out[17]	
	c	pred(c1)|def[11], use@d[12], pred(c2)|def[14]|use, def[16]|use||			
	d	pred(c1)|def[12], use@pred(c2)||			
引数	cond	use@pred(c1)	use@pred(!c1)	def[18]	def[21]
	out				

ステート遷移		ST0->ST1
ローカル変数	a	def[8], use@b[9]
	b	def[9], pred(c1)|use@c[11]|
	c	pred(c1)|def[11], use@d[12], pred(!c2)|def[14]|use, def[16]|use||
	d	pred(c1)|def[12], use@pred(!c2)||
引数	cond	use@pred(c1)
	out	

【図４５】

図４５

ローカル変数削除



【図４６】

図４６

更新後の変数表

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	c	pred(c1)|def[11], pred(c2)|def[16]|use‖		use@out[17]	
引数	cond	use@pred(c1)	use@pred(!c1)		
	out			def[18]	def[21]
ステート遷移		ST0->ST1			
ローカル変数	c	pred(c1)|def[11], pred(!c2)|def[16]|use‖			
引数	cond	use@pred(c1)			
	out				

【図４７】

図４７

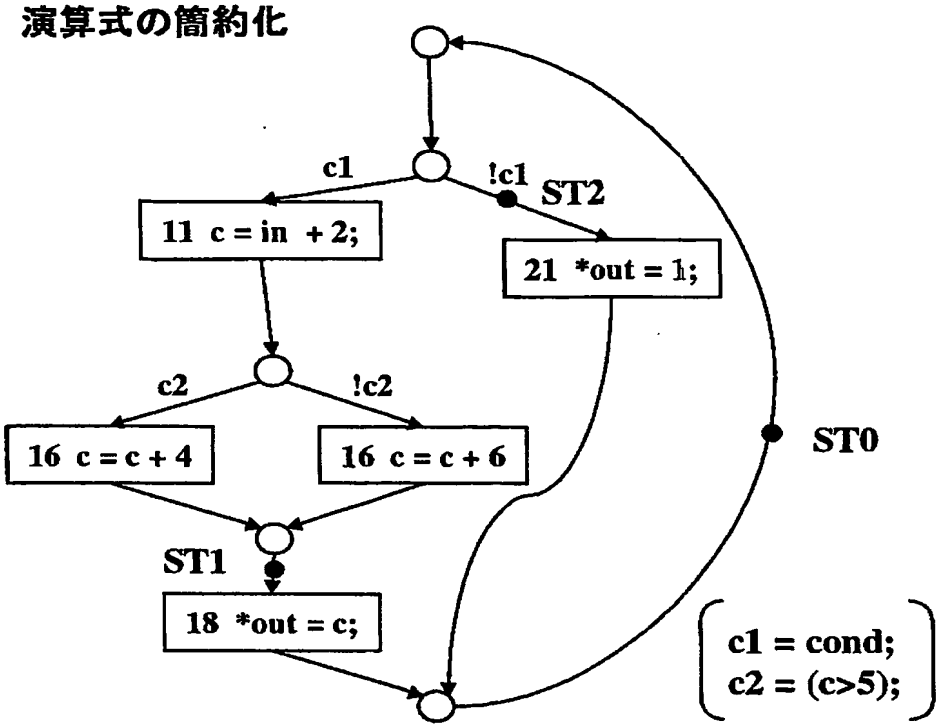
ステート遷移		ST0->ST1	ST0->ST2	ST1->ST0	ST2->ST0
ローカル変数	c	pred(c1)|def[11], pred(c2)|def[16]|use[]	retain	retain	retain
引数	cond	use@pred(c1)	use@pred(!c1)	def[18]	def[21]
	out	retain	retain		
ステート遷移		ST0->ST1			
ローカル変数	c	pred(c1)|def[11], pred(!c2)|def[16]|use[]			
引数	cond	use@pred(c1)			
	out	retain			

〔retain：前置保持〕

【図４８】

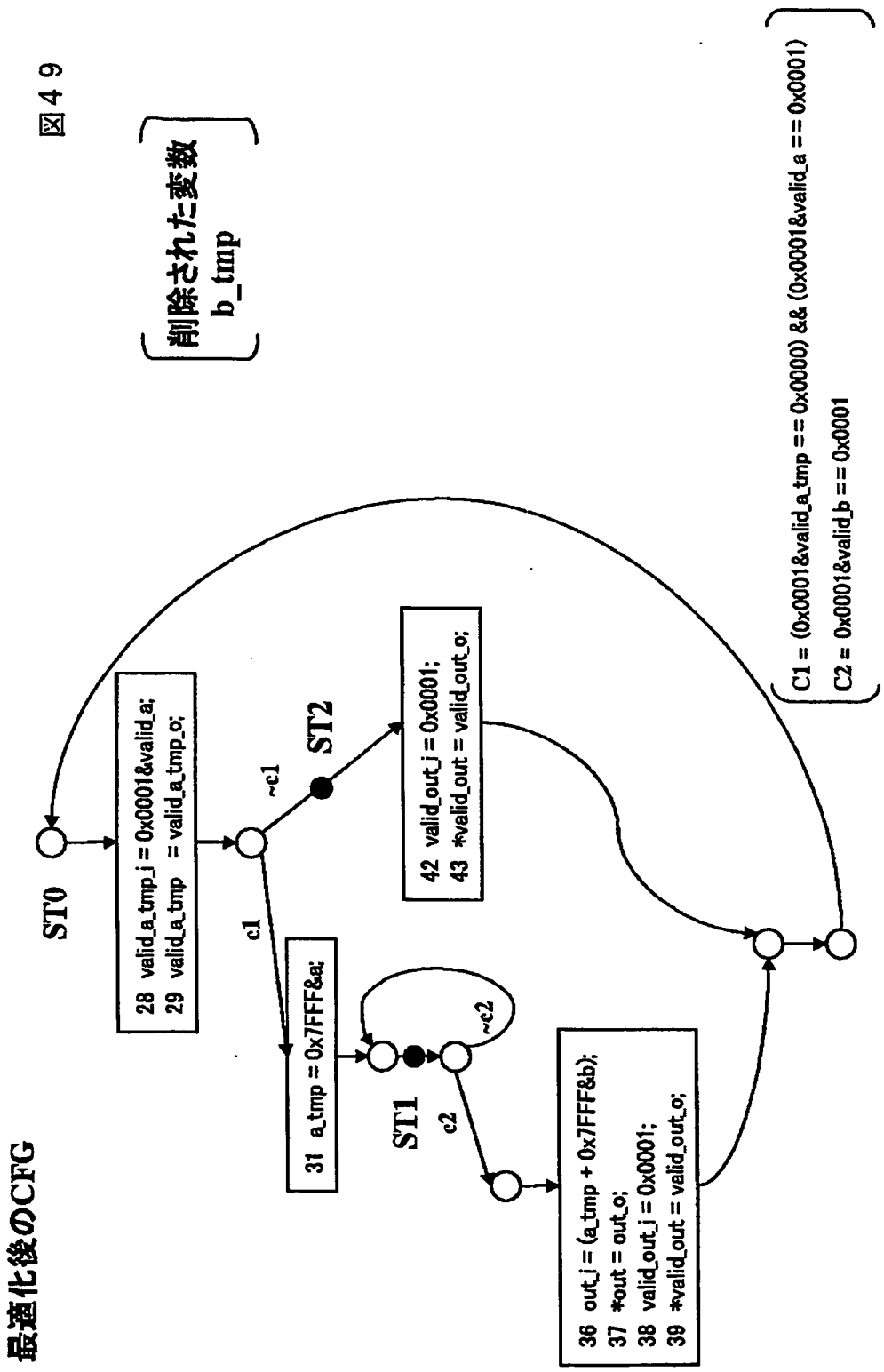
図４８

演算式の簡約化



【図４９】

図４９


最適化後のCFG
ST0
ST1
ST2
c1
~c1
c2
~c2
28 valid_a_tmp_j = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
31 a_tmp = 0x7FFF&a;
42 valid_out_j = 0x0001;
43 *valid_out = valid_out_o;
36 out_j = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_j = 0x0001;
39 *valid_out = valid_out_o;
削除された変数
b_tmp
C1 = (0x0001&valid_a_tmp == 0x0000) && (0x0001&valid_a == 0x0001)
C2 = 0x0001&valid_b == 0x0001

【図５０】

図５０

最適化後の変数表

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST1	ST1->ST1
ローカル変数	valid_a_tmp	def[29], use@pred(c1)	def[29], use@pred(c1)	pred(c2)\def[29], use@pred(c1)|	
	valid_a_tmp_i	def[28]	def[28]	pred(c2)\def[28]|	
	a_tmp	pred(c1)\def[31]|		pred(c2)\pred(c1)\def[31]||	
	out_i			pred(c2)\def[36]|	
	valid_out_i			pred(c2)\def[38]|	
引数	a	pred(c1)\use@a_tmp[31]|		pred(c2)\pred(c1)\use@a_tmp[31]||	
	b			pred(c2)\use@out_i[36]|	
	valid_a	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(c1)|	pred(c2)\use@valid_a_tmp_i[28], use@pred(c1)|	
	valid_b			use@pred(c2)	use@pred(c2)
	valid_out			pred(c2)\def[39]|	
	out_i			pred(c2)\def[37]|	
	valid_a_tmp_o	use@valid_a_tmp[29]	use@valid_a_tmp[29]	pred(c2)\use@valid_a_tmp[29]|	
	valid_out_o			pred(c2)\use@valid_out_o[39]|	
	out_o			pred(c2)\use@out[37]|	
ステート遷移		ST1->ST2	ST2->ST1	ST2->ST2	
ローカル変数	valid_a_tmp	pred(c2)\def[29], use@pred(c1)|	def[29], use@pred(c1)	def[28], use@pred(c1)	
	valid_a_tmp_i	pred(c2)\def[28]|	def[28]	def[28]	
	a_tmp		pred(c1)\def[31]|		
	out_i	pred(c2)\def[36]|			
	valid_out_i	pred(c2)\def[38]|	def[42]	def[42]	
引数	a		pred(c1)\use@a_tmp[31]|		
	b	pred(c2)\use@out_i[36]|			
	valid_a	pred(c2)\use@valid_a_tmp_i[28], use@pred(c1)||	use@valid_a_tmp_i[28], use@pred(c1)	use@valid_a_tmp_i[28], use@pred(c1)	
	valid_b	use@pred(c2)			
	valid_out	pred(c2)\def[39]|	def[43]	def[43]	
	out_i	pred(c2)\def[37]|			
	valid_a_tmp_o	pred(c2)\use@valid_a_tmp[29]|			
	valid_out_o	pred(c2)\use@valid_out_o[39]|	use@valid_out[43]	use@valid_out[43]	
	out_o	pred(c2)\use@out[37]|			

【図５１】

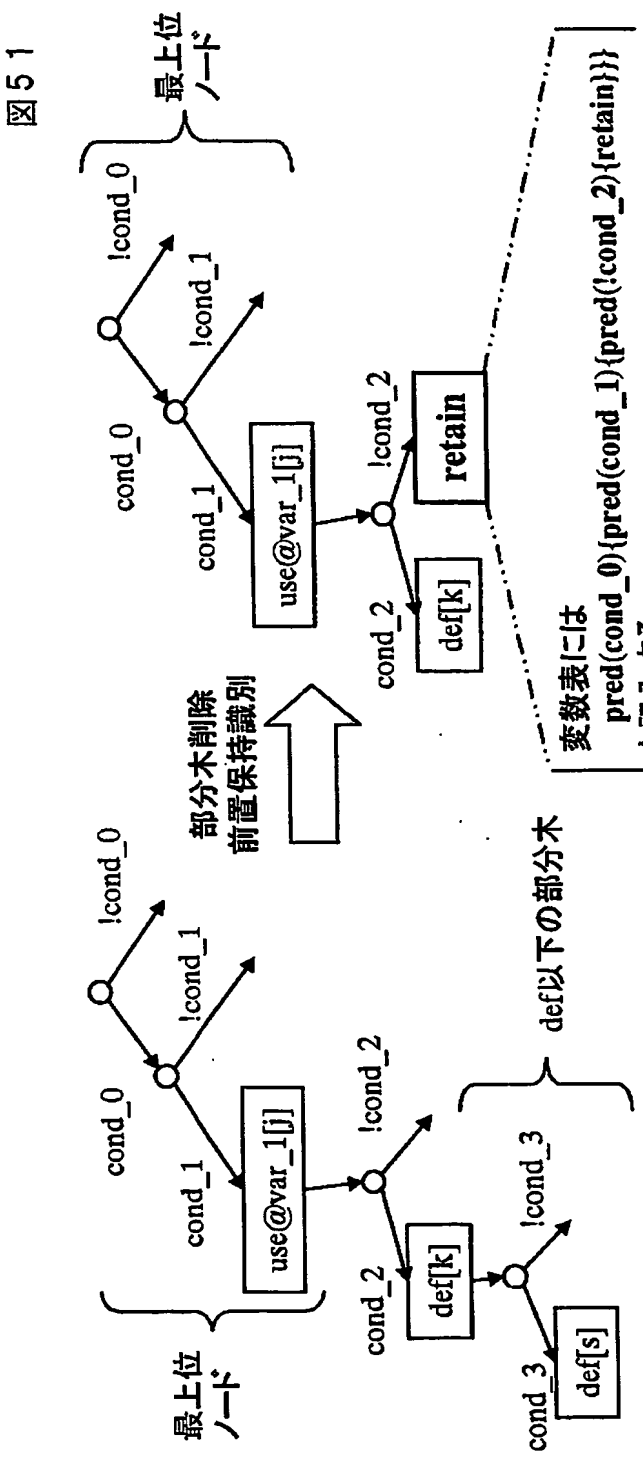


図５１
最上位ノード
!cond_0
!cond_1
cond_0
cond_1
use@var_1[j]
!cond_2
retain
cond_2
def[k]
部分木削除
前置保持識別
変数表には
pred(cond_0){pred(cond_1){pred(!cond_2){retain}}}
と記入する。
!cond_0
!cond_1
cond_0
cond_1
use@var_1[j]
!cond_2
def以下の部分木
cond_2
def[k]
!cond_3
cond_3
def[s]
最上位ノード

【図５２】

図５２

前置保持解析実行後の変数表

ステート遷移		ST0->ST1	ST0->ST2	ST1->ST1	ST1->ST1
ローカル変数	valid_a_tmp	def[29], use●pred(c1)	def[29], use●pred(c1)	pred(c2)|def[29], use●pred(c1)]	pred(c2)|retain]
	valid_a_tmp_i	def[28]	def[28]	pred(c2)|def[28]]	pred(c2)|retain]
	a_tmp	pred(c1)|def[31]]	pred(c1)|retain]	pred(c2)|pred(c1)|def[31]]]	pred(c2)|retain]
	out_i	retain	retain	pred(c2)|def[36]]	pred(c2)|retain]
	valid_out_i	retain	retain	pred(c2)|def[38]]	pred(c2)|retain]
引数	a	pred(c1)|use●a_tmp[31]]		pred(c2)|pred(c1)|use●a_tmp[31]]]	-
	b			pred(c2)|use●out_i[36]]	
	valid_a	use●valid_a_tmp_i[28], use●pred(c1)	use●valid_a_tmp_i[28], use●pred(c1)	pred(c2)|use●valid_a_tmp_i[28], use●pred(c1)]	
	valid_b			use●pred(c2)	use●pred(c2)
	valid_out	retain	retain	pred(c2)|def[39]]	pred(c2)|retain]
	out_i	retain	retain	pred(c2)|def[37]]	pred(c2)|retain]
	valid_a_tmp_o	use●valid_a_tmp[29]	use●valid_a_tmp[29]	pred(c2)|use●valid_a_tmp[29]]	
	valid_out_o			pred(c2)|use●valid_out_o[39]]	
	out_o			pred(c2)|use●out[37]]	
ステート遷移		ST1->ST2	ST2->ST1	ST2->ST2	
ローカル変数	valid_a_tmp	pred(c2)|def[29], use●pred(!c1)]	def[29], use●pred(c1)	def[29], use●pred(!c1)	
	valid_a_tmp_i	pred(c2)|def[28]]	def[28]	def[28]	
	a_tmp	pred(!c1)|retain]	pred(c1)|def[31]]	pred(!c1)|retain]	
	out_i	pred(c2)|def[36]]	retain	retain	
	valid_out_i	pred(c2)|def[38]]	def[42]	def[42]	
引数	a		pred(c1)|use●a_tmp[31]]		
	b	pred(c2)|use●out_i[36]]			
	valid_a	pred(c2)|use●valid_a_tmp_i[28], use●pred(c1)]	use●valid_a_tmp_i[28], use●pred(c1)	use●valid_a_tmp_i[28], use●pred(!c1)	
	valid_b	use●pred(c2)			
	valid_out	pred(c2)|def[39]]	def[43]	def[43]	
	out_i	pred(c2)|def[37]]	retain	retain	
	valid_a_tmp_o	pred(c2)|use●valid_a_tmp[29]]			
	valid_out_o	pred(c2)|use●valid_out_o[39]]	use●valid_out[43]	use●valid_out[43]	
	out_o	pred(c2)|use●out[37]]			

【図５３】

図５３

前置保持解析結果の変数表からの情報取得

ステート遷移	ST0->ST1	ST0->ST2	ST1->ST1	ST1->ST1

ローカル変数	valid_a_tmp
	valid_a_tmp_i
	a_tmp
	out_i
	valid_out_i
		def[29], use@pred(c1)
		def[28]
		pred(c1)\def[31]
		out_i = out_o;
		valid_out_i = valid_out_o;
			def[29], use@pred(c1)
			def[28]
			pred(c1)\nxt_a_tmp = tmp;
			out_i = out_o;
			valid_out_i = valid_out_o;
				pred(c2)\valid_a_tmp = valid_a_tmp_o;
				pred(c2)\valid_a_tmp_i = valid_a_tmp_o;
				pred(c2)\nxt_a_tmp = a_tmp;
				pred(c2)\out_i = out_o;
				pred(c2)\valid_out_i = valid_out_o;

引数	a
	b
	valid_a
	valid_b
	valid_out
	out
	valid_a_tmp_o
	valid_out_o
	out_o
		pred(c1)\use@a_tmp[31]
		use@valid_a_tmp_i[28], use@pred(c1)
		valid_out = valid_out_o;
		out = out_o;
		use@valid_a_tmp[29]
			pred(c2)\pred(c1)\use@a_tmp[31]
			pred(c2)\use@out_i[36]
			pred(c2)\use@valid_a_tmp_i[28], use@pred(c1)
			use@pred(c2)
			pred(c2)\def[39]
			pred(c2)\def[37]
			pred(c2)\use@valid_a_tmp[29]
			pred(c2)\use@valid_out_o[39]
			pred(c2)\use@out[37]
				use@pred(c2)
				pred(c2)\valid_out = valid_out_o;
				pred(c2)\out = out_o;

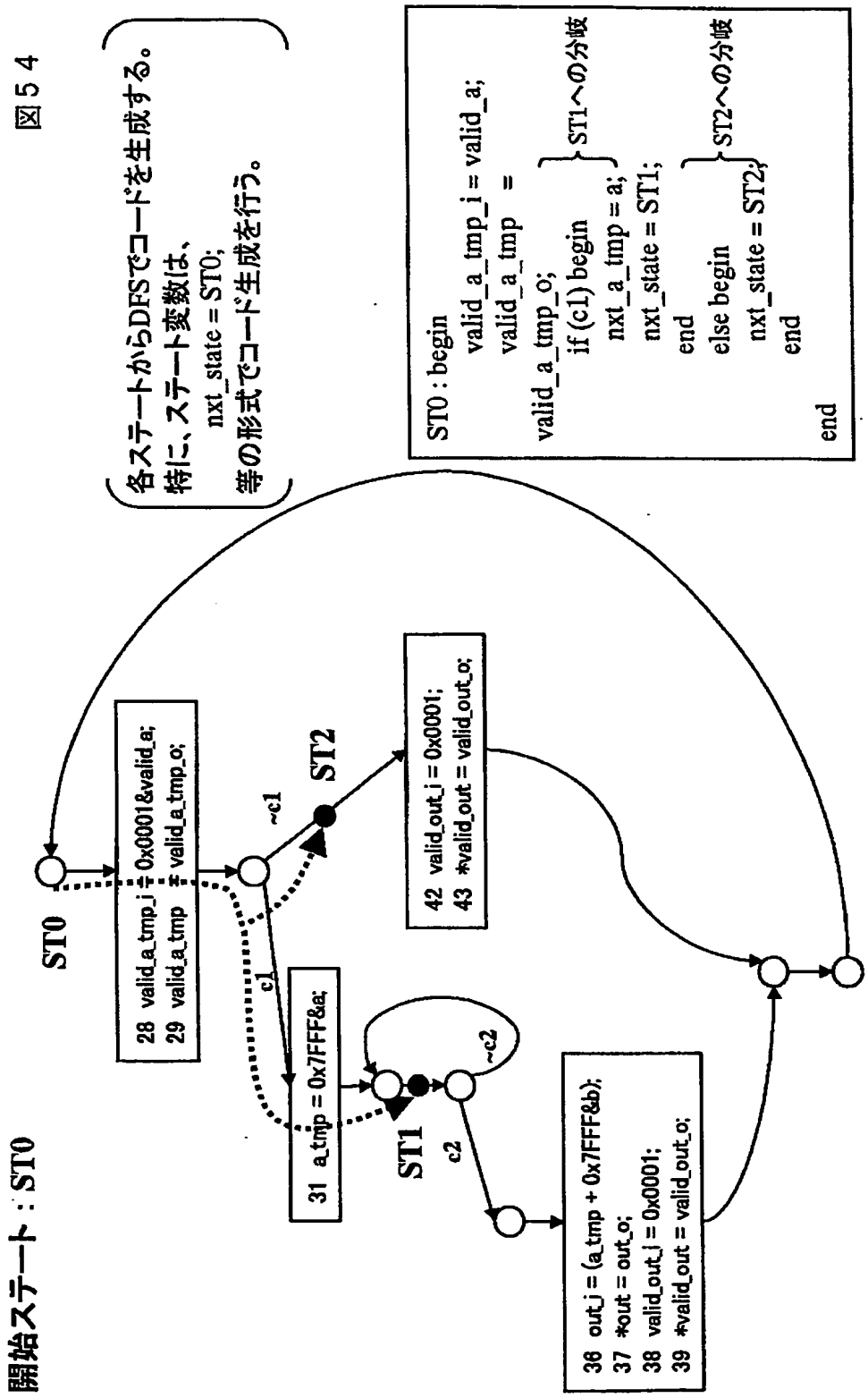
ステート遷移	ST1->ST2	ST2->ST1	ST2->ST2	

ローカル変数	valid_a_tmp
	valid_a_tmp_i
	a_tmp
	out_i
	valid_out_i
		pred(c2)\def[29], use@pred(c1)\)
		pred(c2)\def[28])
		pred(c1)\nxt_a_tmp = a_tmp;
		pred(c2)\def[36]
		pred(c2)\def[38]
			def[29], use@pred(c1)
			def[28]
			pred(c1)\def[31])
			out_i = out_o;
			def[42]
				def[29], use@pred(c1)
				def[28]
				pred(c1)\nxt_a_tmp = a_tmp;
				out_i = out_o;
				def[42]

引数	a b valid_a valid_b valid_out out valid_a_tmp_o valid_out_o out_o	pred(c2)\use@out_i[36] pred(c2)\use@valid_a_tmp_i[28], use@pred(c1)) use@pred(c2) pred(c2)\def[39]) pred(c2)\def[37]) pred(c2)\use@valid_a_tmp[29]) pred(c2)\use@valid_out_o[39]) pred(c2)\use@out[37])	pred(c1)\use@a_tmp[31]) use@valid_a_tmp_i[28], use@pred(c1) def[43] out = out_o; use@valid_out[43]	use@valid_a_tmp_i[28], use@pred(c1) def[43] out = out_o; use@valid_out[43]

【図５４】

図５４


各ステートからDFSでコードを生成する。
特に、ステート変数は、
nxt_state = ST0;
等の形式でコード生成を行う。
ST0 : begin
valid_a_tmp_i = valid_a;
valid_a_tmp =
valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_state = ST2;
end
end
ST1へ の分岐
ST2へ の分岐
開始ステート：ST0
ST0
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
c1
~c1
ST2
31 a_tmp = 0x7FFF&a;
ST1
c2
~c2
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
36 out_i = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;

【図５５】

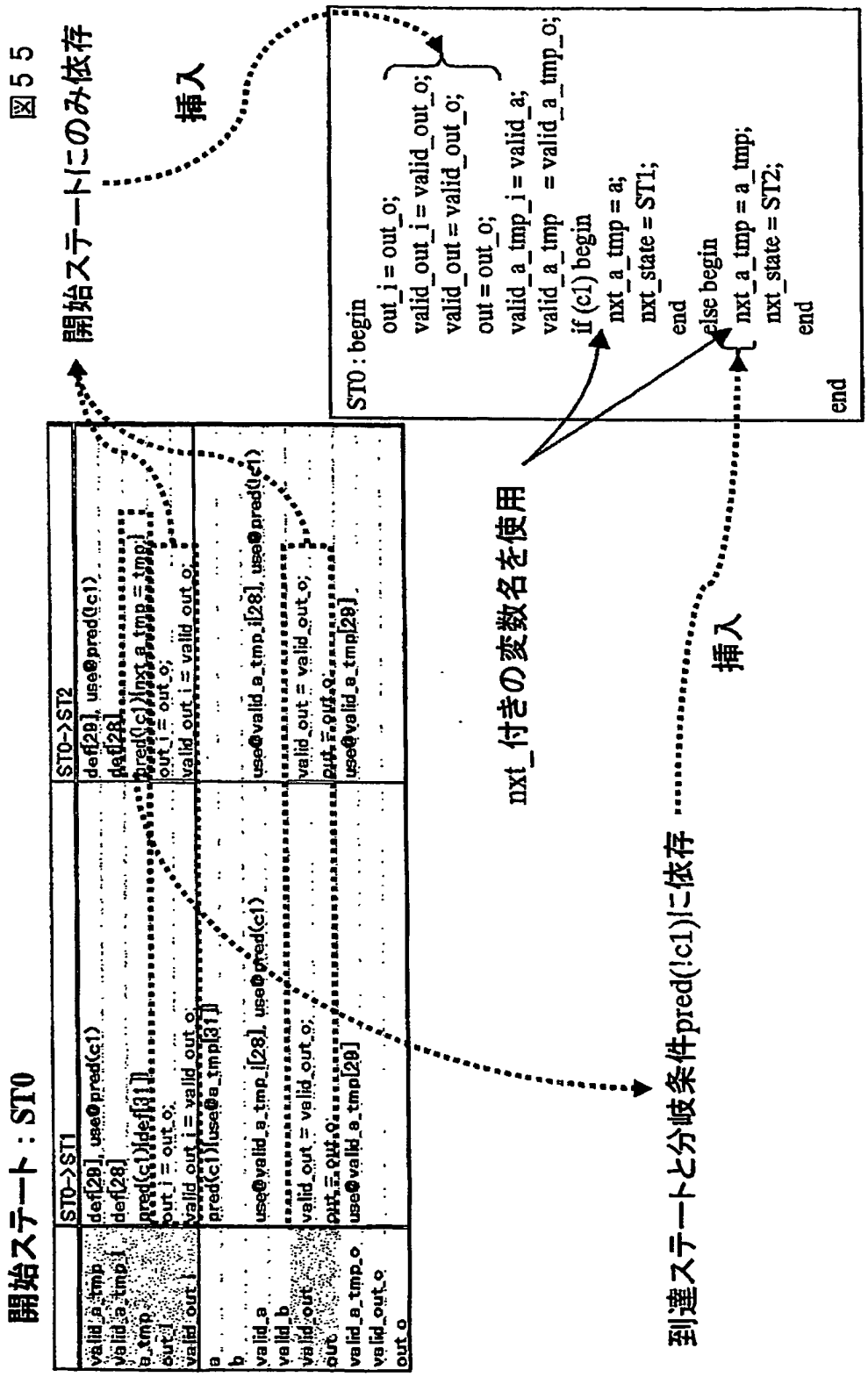


図５５
開始ステート：ST0
ST0->ST1
ST0->ST2
valid_a_tmp
valid_a_tmp_i
a_tmp
out_i
valid_out_i
a
b
valid_a
valid_b
valid_out
out_o
valid_a_tmp_o
valid_out_o
def[28], use@pred(c1)
def[28]
pred(c1)/def[31]
out_i = out_o;
valid_out_i = valid_out_o;
pred(c1)/use@a_tmp[31]
use@valid_a_tmp_i[28], use@pred(c1)
valid_out = valid_out_o;
use@valid_a_tmp[29]
def[28], use@pred(c1)
def[28]
pred(c1)/def[31]
out_i = out_o;
valid_out_i = valid_out_o;
use@valid_a_tmp_i[28], use@pred(!c1)
valid_out = valid_out_o;
use@valid_a_tmp[29]
開始ステートにのみ依存
挿入
ST0 : begin
out_i = out_o;
valid_out_i = valid_out_o;
valid_out = valid_out_o;
out = out_o;
valid_a_tmp_i = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_a_tmp = a_tmp;
nxt_state = ST2;
end
end
nxt_付きの変数名を使用
到達ステートと分岐条件pred(!c1)に依存
挿入

【図５６】

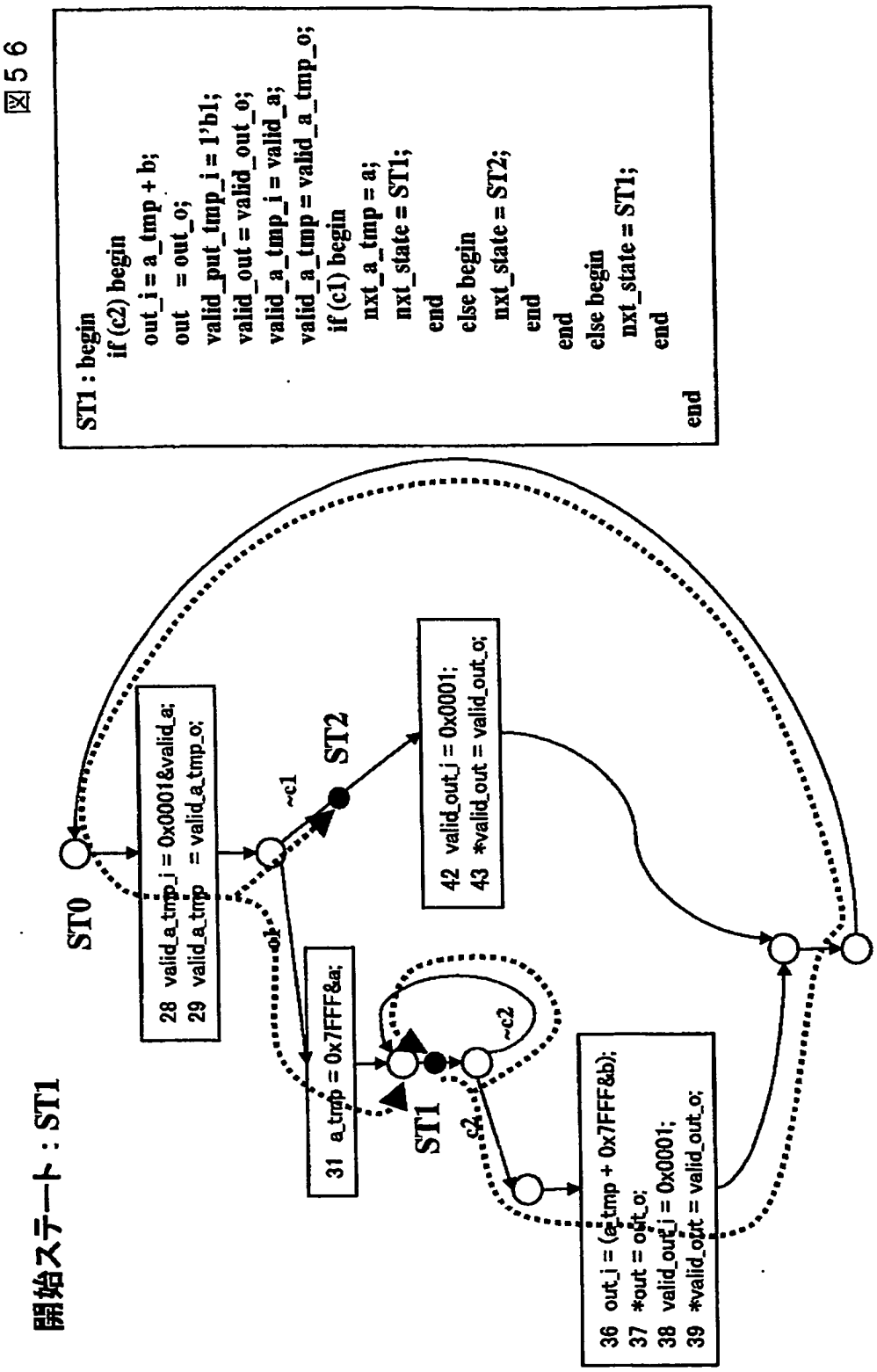


図５６
ST1 : begin
if (c2) begin
out_i = a_tmp + b;
out = out_o;
valid_put_tmp_i = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_i = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_state = ST2;
end
end
else begin
nxt_state = ST1;
end
end
開始ステート：ST1
ST0
ST1
ST2
~c1
c1
~c2
c2
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp = valid_a_tmp_o;
31 a_tmp = 0x7FFF&a;
42 valid_out_i = 0x0001;
43 *valid_out = valid_out_o;
36 out_i = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;

【図５７】

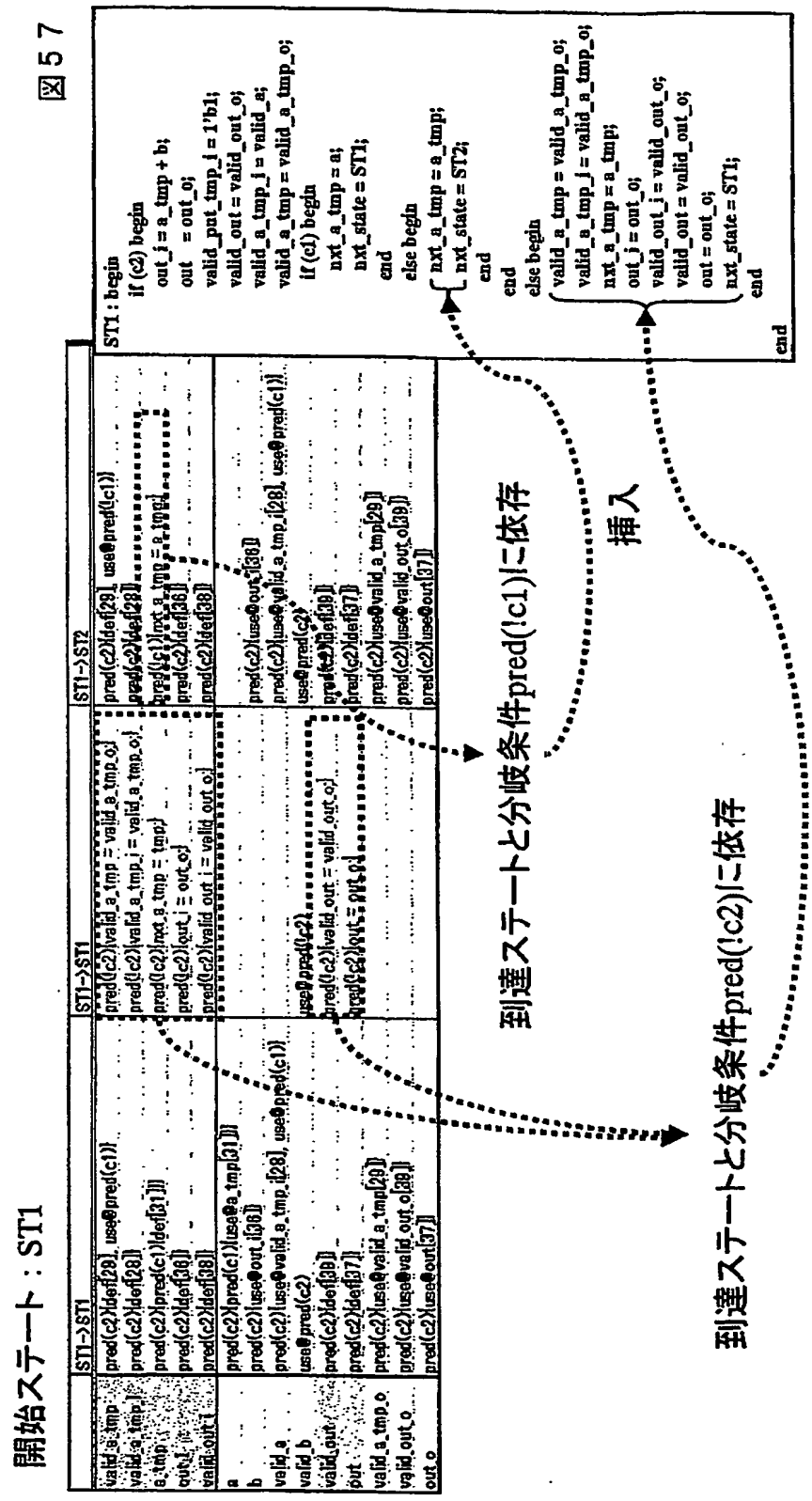
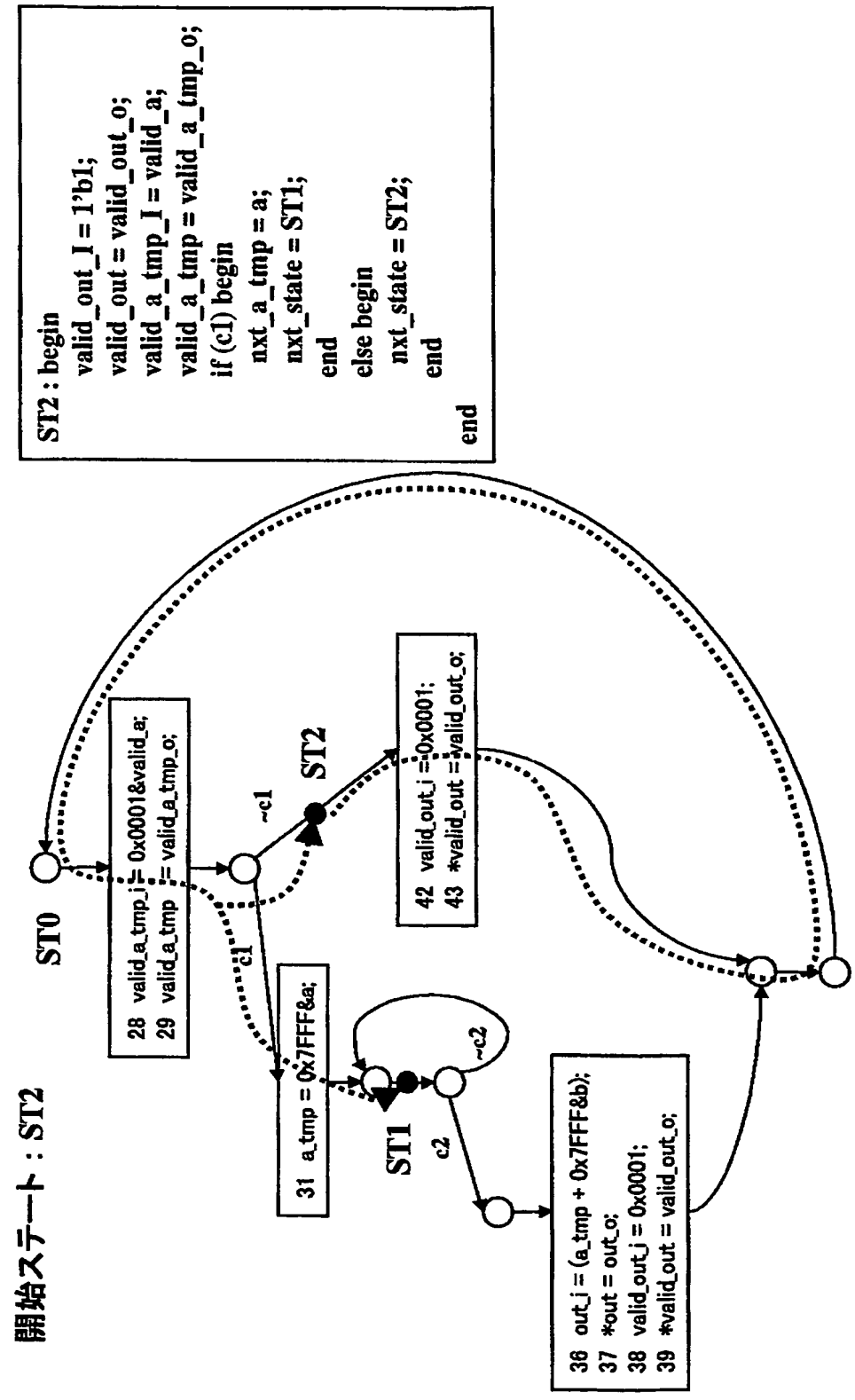


図５７
開始ステート：ST1
ST1->ST1
ST1->ST2
valid_a_tmp
valid_a_tmp_i
a_tmp
out_i
valid_out_i
a
b
valid_a
valid_b
valid_out
out
valid_a_tmp_o
valid_out_o
out_o
pred(c2)\def[28] use@pred(c1)
pred(!c2)\valid_a_tmp = valid_a_tmp_o;
pred(!c2)\valid_a_tmp_i = valid_a_tmp_o;
pred(!c2)\nxt_a_tmp = tmp;
pred(!c2)\out_i = out_o;
pred(!c2)\valid_out_i = valid_out_o;
pred(c2)\def[28]
pred(c2)\def[31]
pred(c1)\use@a_tmp[31]
pred(c2)\use@out_i[36]
pred(c2)\use@valid_a_tmp_i[28], use@pred(c1)
use@pred(c2)
pred(c2)\def[39]
pred(c2)\valid_out = valid_out_o;
pred(c2)\def[37]
pred(c2)\use@valid_a_tmp[29]
pred(c2)\use@valid_out_o[39]
pred(c2)\use@out[37]
pred(c2)\def[36]
pred(c2)\def[38]
pred(c2)\use@out_i[38]
pred(c2)\use@valid_a_tmp_i[28], use@pred(c1)
pred(c2)\def[39]
pred(c2)\def[37]
pred(c2)\use@valid_a_tmp[29]
pred(c2)\use@valid_out_o[39]
pred(c2)\use@out[37]
ST1 : begin
if (c2) begin
out_i = a_tmp + b;
out = out_o;
valid_out_tmp_i = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_i = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_a_tmp = a_tmp;
nxt_state = ST2;
end
end
else begin
valid_a_tmp = valid_a_tmp_o;
valid_a_tmp_i = valid_a_tmp_o;
nxt_a_tmp = a_tmp;
out_i = out_o;
valid_out_i = valid_out_o;
valid_out = valid_out_o;
out = out_o;
nxt_state = ST1;
end
end
到達ステートと分岐条件pred(!c1)に依存
挿入
到達ステートと分岐条件pred(!c2)に依存
挿入

【図５８】

図５８


ST2 : begin
valid_out_I = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_I = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_state = ST2;
end
end
開始ステート：ST2
ST0
28 valid_a_tmp_i = 0x0001&valid_a;
29 valid_a_tmp := valid_a_tmp_o;
c1
~c1
ST2
42 valid_out_i =0x0001;
43 *valid_out =valid_out_o;
31 a_tmp = 0x7FFF&a;
ST1
c2
~c2
36 out_i = (a_tmp + 0x7FFF&b);
37 *out = out_o;
38 valid_out_i = 0x0001;
39 *valid_out = valid_out_o;

【図５９】

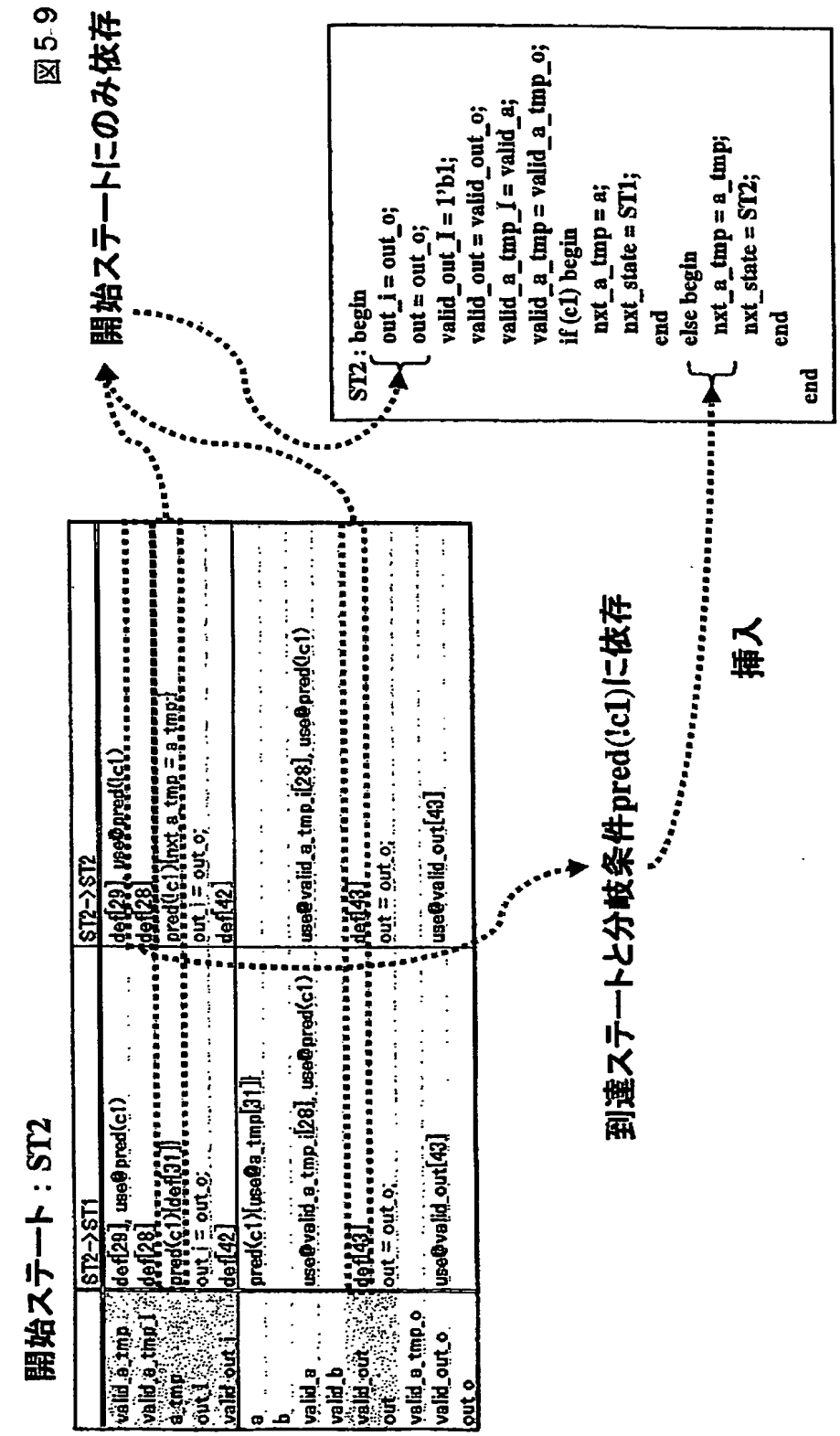


図5-9
開始ステート：ST2
ST2->ST1
ST2->ST2
valid_a_tmp
valid_a_tmp_i
a_tmp
out_i
valid_out_i
def[29] use@pred(c1)
def[28]
pred(c1)def[31]
out_i = out_o;
def[42]
pred(c1)use@a_tmp[31]
def[43]
use@valid_a_tmp_i[28] use@pred(c1)
a
b
valid_a
valid_b
valid_out
out
valid_a_tmp_o
valid_out_o
out_o
def[29] use@pred(c1)
def[28]
pred(c1)nxt_a_tmp = a_tmp;
out_i = out_o;
def[42]
def[43]
out = out_o;
use@valid_out[43]
開始ステートにのみ依存
到達ステートと分岐条件pred(!c1)に依存
挿入
ST2 : begin
out_i = out_o;
out = out_o;
valid_out_I = 1'b1;
valid_out = valid_out_o;
valid_a_tmp_I = valid_a;
valid_a_tmp = valid_a_tmp_o;
if (c1) begin
nxt_a_tmp = a;
nxt_state = ST1;
end
else begin
nxt_a_tmp = a_tmp;
nxt_state = ST2;
end
end

【図６０】

図６０

```
1  module PipeLine(clk, reset_n,
2                  valid_a, valid_b, a, b,
3                  out, valid_out);
   // System clock and reset
4  input clk;
5  input reset_n;
   // PipeLine input signals
6  input valid_a;
7  input valid_b;
8  input [14:0] a;
9  input [14:0] b;
   // PipeLine output signals
10 output valid_out;
11 reg valid_out;
12 output [15:0] out;
13 reg [15:0] out;
   // PipeLine internal signals
14 reg valid_a_tmp;
15 reg valid_a_tmp_i;
16 reg valid_a_tmp_o;
17 reg [14:0] a_tmp;
18 reg [14:0] nxt_a_tmp;
19 reg valid_out_i;
20 reg valid_out_o;
21 reg [15:0] out_i;
22 reg [15:0] out_o;
   // State registers
23 reg [1:0] state, nxt_state;
24 parameter ST0=2'b00,
25           ST1=2'b01,
26           ST2=2'b10;
   // Blanch conditions
27 wire c1;
28 wire c2;
29 assign c1 = !valid_a_tmp&&valid_a;
30 assign c2 = valid_b;
```

```
    // Mealy finite state machine
51  always @ (state or c1 or c2 or
52              valid_a_tmp_i or valid_a_tmp_o or
53              valid_a_tmp or a_tmp or
54              valid_out_i or valid_out_o or
55              out_i or out_o) begin
56    case(state[1:0])
57    ST0 : begin
58          valid_a_tmp_i   = valid_a;
59          valid_a_tmp = valid_a_tmp_o;
60          valid_out_i = valid_out_o;
61          valid_out       = valid_out_o;
62          out_i           = out_o;
63          out             = out_o;
64          if (c1) begin
65            nxt_a_tmp = a;
66            nxt_state = ST1;
67          end
68          else begin
69            nxt_a_tmp = a_tmp;
70            nxt_state = ST2;
71          end
72    end
```

```
    // Regsiter assignment statement
31  always @ (posedge clk or negedge reset_n) begin
32    if (!reset_n) begin
33      valid_a_tmp_o <= 1'b0;
34      out_o         <= 17'b00000000000000000;
35    end
36    else begin
37      valid_a_tmp_o <= valid_a_tmp_i;
38      out_o         <= out_i;
39    end
40  end
    // State registers and temporal registers
41  always @ (posedge clk or negedge reset_n) begin
42    if (!reset_n) begin
43      state         <= ST0;
44      a_tmp         <= 16'b0;
45    end
46    else begin
47      state         <= nxt_state;
48      a_tmp         <= nxt_a_tmp;
49    end
50  end
```

【図６２】

図６２

```
ST1 : begin
      if (c2) begin
        out_j       = a_tmp + b;
        out         = out_o;
        valid_out_j = 1'b1;
        valid_out   = valid_out_o;
        valid_a_tmp_j  = valid_a;
        valid_a_tmp = valid_a_tmp_o;
        if (c1) begin
          nxt_a_tmp = a;
          nxt_state = ST1;
        end
        else begin
          nxt_a_tmp = a_tmp;
          nxt_state = ST2;
        end
      end
      else begin
        nxt_state   = ST1;
        valid_a_tmp_j  = valid_a_tmp_o;
        valid_a_tmp = valid_a_tmp_o;
        nxt_a_tmp   = a_tmp;
        valid_out_j = valid_out_o;
        valid_out   = valid_out_o;
        out_j       = out_o;
        out         = out_o;
      end
    end
```

```
ST2 : begin
        valid_a_tmp_j  = valid_a;
        valid_a_tmp = valid_a_tmp_o;
        valid_out_j = 1'b0;
        valid_out   = valid_out_o;
        out_j       = out_o;
        out         = out_o;
        if (c1) begin
          nxt_a_tmp = a;
          nxt_state = ST1;
        end
        else begin
          nxt_a_tmp = a_tmp;
          nxt_state = ST2;
        end
    end
    default : begin
      nxt_state   = ST0;
      valid_a_tmp_j  = valid_a_tmp_o;
      valid_a_tmp = 1'b0;
      nxt_a_tmp   = 15'b0;
      valid_out_j = valid_out_o;
      valid_out   = 1'b0;
      out_j       = out_o;
      out         = out_o;
    end
  endcase
end
endmodule
```

【書類名】　要約書

【要約】

【課題】　ストール動作を伴うパイプライン動作が可能な回路のプログラム記述又は回路記述を容易に得る事ができるコンパイラを提供する。

【解決手段】　クロック境界及びレジスタ代入文によりステートメントレベルでの並列動作の記述をサイクル精度で記述可能な擬似Ｃ記述（１）を入力とし、レジスタ代入文の識別を行い（Ｓ２）、実行可能なＣ記述（３）を生成すると共に、状態数削減を行ったステートマシンを抽出し、０サイクルで実行されるループが存在するか否かを判定し（Ｓ５）、もしなければ、論理合成可能な回路記述（４）を生成する。クロック境界を明示的にＣ記述内に挿入した擬似Ｃ記述を入力し、レジスタ代入文によるステートメントレベルでの並列記述を可能にした擬似Ｃ記述を入力するから、ストール動作を伴うパイプライン動作が表現可能である。

【選択図】　図１

【書類名】　出願人名義変更届（一般承継）
【あて先】　特許庁長官　殿
【事件の表示】
　【出願番号】　特願2002-300073
【承継人】
　【識別番号】　503121103
　【氏名又は名称】　株式会社ルネサステクノロジ
【承継人代理人】
　【識別番号】　100089071
　【弁理士】
　【氏名又は名称】　玉村　静世
【提出物件の目録】
　【包括委任状番号】　0308734
　【物件名】　承継人であることを証明する登記簿謄本　1
　　【援用の表示】　特許第３１５４５４２号　平成１５年４月１１日付け提出の会社分割による特許権移転登録申請書　を援用する
　【物件名】　権利の承継を証明する承継証明書　1
　　【援用の表示】　特願平２－３２１６４９号　同日提出の出願人名義変更届（一般承継）を援用する
【プルーフの要否】　要

認定・付加情報

特許出願の番号	特願２００２－３００７３
受付番号	５０３０１２１０８０９
書類名	出願人名義変更届（一般承継）
担当官	末武　実　　１９１２
作成日	平成１５年１０月　７日

＜認定情報・付加情報＞

【提出日】　平成15年 7月23日

特願２００２－３００７３

出　願　人　履　歴　情　報

識別番号　　［０００００５１０８］

１．変更年月日　　１９９０年　８月３１日
［変更理由］　　新規登録
住　所　　東京都千代田区神田駿河台４丁目６番地
氏　名　　株式会社日立製作所

特願２００２－３００７３

出　願　人　履　歴　情　報

識別番号　　［５０３１２１１０３］

１．変更年月日　　２００３年　４月　１日
［変更理由］　　新規登録
住　所　　東京都千代田区丸の内二丁目４番１号
氏　名　　株式会社ルネサステクノロジ

Translation

PATENT COOPERATION TREATY

PCT/JP2003/012839

PCT

INTERNATIONAL PRELIMINARY EXAMINATION REPORT

(PCT Article 36 and Rule 70)

Applicant's or agent's file reference 310201409WO1	FOR FURTHER ACTION See Notification of Transmittal of International Preliminary Examination Report (Form PCT/IPEA/416)	
International application No. PCT/JP2003/012839	International filing date (*day/month/year*) 07 October 2003 (07.10.2003)	Priority date (*day/month/year*) 15 October 2002 (15.10.2002)
International Patent Classification (IPC) or national classification and IPC G06F 17/50		
Applicant RENESAS TECHNOLOGY CORP.		

1. This international preliminary examination report has been prepared by this International Preliminary Examining Authority and is transmitted to the applicant according to Article 36.

2. This REPORT consists of a total of 4 sheets, including this cover sheet.

   ☒ This report is also accompanied by ANNEXES, i.e., sheets of the description, claims and/or drawings which have been amended and are the basis for this report and/or sheets containing rectifications made before this Authority (see Rule 70.16 and Section 607 of the Administrative Instructions under the PCT).

   These annexes consist of a total of 3 sheets.

3. This report contains indications relating to the following items:

   - I ☒ Basis of the report
   - II ☐ Priority
   - III ☐ Non-establishment of opinion with regard to novelty, inventive step and industrial applicability
   - IV ☐ Lack of unity of invention
   - V ☒ Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement
   - VI ☐ Certain documents cited
   - VII ☐ Certain defects in the international application
   - VIII ☒ Certain observations on the international application

Date of submission of the demand	Date of completion of this report
07 October 2003 (07.10.2003)	27 January 2004 (27.01.2004)
Name and mailing address of the IPEA/JP Facsimile No.	Authorized officer Telephone No.

INTERNATIONAL PRELIMINARY EXAMINATION REPORT

International application No.
PCT/JP2003/012839

I. Basis of the report

1. With regard to the **elements** of the international application:*

☐ the international application as originally filed

☒ the description:

pages 1-35, as originally filed
pages _____, filed with the demand
pages _____, filed with the letter of _____

☒ the claims:

pages 4,5, 7-13, 15-20, as originally filed
pages _____, as amended (together with any statement under Article 19
pages _____, filed with the demand
pages 1-3, 6, 14, filed with the letter of 19 January 2004 (19.01.2004)

☒ the drawings:

pages _____, as originally filed
pages _____, filed with the demand
pages _____, filed with the letter of _____

☐ the sequence listing part of the description:

pages _____, as originally filed
pages _____, filed with the demand
pages _____, filed with the letter of _____

2. With regard to the **language**, all the elements marked above were available or furnished to this Authority in the language in which the international application was filed, unless otherwise indicated under this item.
These elements were available or furnished to this Authority in the following language _____ which is:

☐ the language of a translation furnished for the purposes of international search (under Rule 23.1(b)).
☐ the language of publication of the international application (under Rule 48.3(b)).
☐ the language of the translation furnished for the purposes of international preliminary examination (under Rule 55.2 and/or 55.3).

3. With regard to any **nucleotide and/or amino acid sequence** disclosed in the international application, the international preliminary examination was carried out on the basis of the sequence listing:

☐ contained in the international application in written form.
☐ filed together with the international application in computer readable form.
☐ furnished subsequently to this Authority in written form.
☐ furnished subsequently to this Authority in computer readable form.
☐ The statement that the subsequently furnished written sequence listing does not go beyond the disclosure in the international application as filed has been furnished.
☐ The statement that the information recorded in computer readable form is identical to the written sequence listing has been furnished.

4. ☐ The amendments have resulted in the cancellation of:

☐ the description, pages _____
☐ the claims, Nos. _____
☐ the drawings, sheets/fig _____

5. ☐ This report has been established as if (some of) the amendments had not been made, since they have been considered to go beyond the disclosure as filed, as indicated in the Supplemental Box (Rule 70.2(c)).**

* *Replacement sheets which have been furnished to the receiving Office in response to an invitation under Article 14 are referred to in this report as "originally filed" and are not annexed to this report since they do not contain amendments (Rule 70.16 and 70.17).*

** *Any replacement sheet containing such amendments must be referred to under item 1 and annexed to this report.*

INTERNATIONAL PRELIMINARY EXAMINATION REPORT

International application No.
PCT/JP03/12839

V. Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

1. Statement

Novelty (N)	Claims	2, 3, 7-20	YES
	Claims	1, 4-6	NO
Inventive step (IS)	Claims	15-20	YES
	Claims	1-14	NO
Industrial applicability (IA)	Claims	1-20	YES
	Claims		NO

2. Citations and explanations

Document 1: KAZUTOSHI WAKABAYASHI, ET AL., "Densoyo LSI o Dosa Gosei de Kaihatsu, Kino Sekkei no Kikan ga 1/10 ni Tanshuku," Nikkei Electronics, Nikkei Business Publications, Inc., 12 February 1996 (12.02.96), No. 655, pages 147-169

Document 2: JP, 2002-49652, A (HIROSHI YASUDA), February 15, 2002 (02.15. 02.02), claims 1-5 (Family: none)

The Cyber described in document 1 performs high-level synthesis based on the operation description language BDL (with grammar similar to C language) and outputs RTL description. The aforesaid BDL can describe by changing the clock change point (i.e. the clock boundary) to"$".

Here, when inserting to a variable between $ and $, it is unclear to what register the variable is allocated, but insertion to a register occurs, so it could be called a register insertion statement.

Furthermore, the "=" and "++" in document 1 could be called "specific operators."

Also, document 2 discloses something equivalent to a second program description.

Meanwhile, compiling something described in one language to a description using another language is well-known art.

This being the case, compiling the aforesaid BDL to the description described in document 2 is merely something to be appropriately achieved by a person skilled in the art.

VIII. Certain observations on the international application

The following observations on the clarity of the claims, description, and drawings or on the question whether the claims are fully supported by the description, are made:

The technical scope of "register insertion statement" in claims 1 through 20 is unclear.

特　許　協　力　条　約

PCT

国際予備審査報告

(法第１２条、法施行規則第56条)
〔ＰＣＴ３６条及びＰＣＴ規則70〕

出願人又は代理人の書類記号　３１０２０１４０９ＷＯ１	今後の手続きについては、国際予備審査報告の送付通知（様式ＰＣＴ／ＩＰＥＡ／４１６）を参照すること。	
国際出願番号 ＰＣＴ／ＪＰ０３／１２８３９	国際出願日 (日.月.年)　０７．１０．０３	優先日 (日.月.年)　１５．１０．０２
国際特許分類（ＩＰＣ） Ｉｎｔ．Ｃｌ．⁷　Ｇ０６Ｆ１７／５０		
出願人（氏名又は名称） 株式会社ルネサステクノロジ		

1．　国際予備審査機関が作成したこの国際予備審査報告を法施行規則第57条（ＰＣＴ３６条）の規定に従い送付する。

2．　この国際予備審査報告は、この表紙を含めて全部で　4　ページからなる。

☒ この国際予備審査報告には、附属書類、つまり補正されて、この報告の基礎とされた及び／又はこの国際予備審査機関に対してした訂正を含む明細書、請求の範囲及び／又は図面も添付されている。
（ＰＣＴ規則70.16及びＰＣＴ実施細則第６０７号参照）
この附属書類は、全部で　3　ページである。

3．　この国際予備審査報告は、次の内容を含む。

I　☒　国際予備審査報告の基礎

II　☐　優先権

III　☐　新規性、進歩性又は産業上の利用可能性についての国際予備審査報告の不作成

IV　☐　発明の単一性の欠如

V　☒　ＰＣＴ３５条(2)に規定する新規性、進歩性又は産業上の利用可能性についての見解、それを裏付けるための文献及び説明

VI　☐　ある種の引用文献

VII　☐　国際出願の不備

VIII　☒　国際出願に対する意見

国際予備審査の請求書を受理した日 ０７．１０．０３	国際予備審査報告を作成した日 ２７．０１．０４
名称及びあて先 日本国特許庁（ＩＰＥＡ／ＪＰ） 郵便番号１００－８９１５ 東京都千代田区霞が関三丁目４番３号	特許庁審査官（権限のある職員）　５Ｈ　９６５２ 早川　学 電話番号　０３－３５８１－１１０１　内線　３５３１

様式ＰＣＴ／ＩＰＥＡ／４０９（表紙）（１９９８年７月）

国際予備審査報告

国際出願番号　ＰＣＴ／ＪＰ０３／１２８３９

Ⅰ．　国際予備審査報告の基礎

1．　この国際予備審査報告は下記の出願書類に基づいて作成された。（法第６条（ＰＣＴ１４条）の規定に基づく命令に応答するために提出された差し替え用紙は、この報告書において「出願時」とし、本報告書には添付しない。ＰＣＴ規則70.16, 70.17）

☐ 出願時の国際出願書類

☒ 明細書　第 1-35 ページ、　出願時に提出されたもの
明細書　第 　　ページ、　国際予備審査の請求書と共に提出されたもの
明細書　第 　　ページ、　　付の書簡と共に提出されたもの

☒ 請求の範囲　第 4, 5, 7-13, 15-20 項、　出願時に提出されたもの
請求の範囲　第 　　項、　ＰＣＴ１９条の規定に基づき補正されたもの
請求の範囲　第 　　項、　国際予備審査の請求書と共に提出されたもの
請求の範囲　第 1-3, 6, 14 項、　１９．０１．０４　付の書簡と共に提出されたもの

☒ 図面　第 1-62 ~~ページ~~図、　出願時に提出されたもの
図面　第 　　ページ／図、　国際予備審査の請求書と共に提出されたもの
図面　第 　　ページ／図、　　付の書簡と共に提出されたもの

☐ 明細書の配列表の部分　第 　　ページ、　出願時に提出されたもの
明細書の配列表の部分　第 　　ページ、　国際予備審査の請求書と共に提出されたもの
明細書の配列表の部分　第 　　ページ、　　付の書簡と共に提出されたもの

2．　上記の出願書類の言語は、下記に示す場合を除くほか、この国際出願の言語である。

上記の書類は、下記の言語である 　　語である。

☐ 国際調査のために提出されたＰＣＴ規則23.1(b)にいう翻訳文の言語
☐ ＰＣＴ規則48.3(b)にいう国際公開の言語
☐ 国際予備審査のために提出されたＰＣＴ規則55.2または55.3にいう翻訳文の言語

3．　この国際出願は、ヌクレオチド又はアミノ酸配列を含んでおり、次の配列表に基づき国際予備審査報告を行った。

☐ この国際出願に含まれる書面による配列表
☐ この国際出願と共に提出された磁気ディスクによる配列表
☐ 出願後に、この国際予備審査（または調査）機関に提出された書面による配列表
☐ 出願後に、この国際予備審査（または調査）機関に提出された磁気ディスクによる配列表
☐ 出願後に提出した書面による配列表が出願時における国際出願の開示の範囲を超える事項を含まない旨の陳述書の提出があった
☐ 書面による配列表に記載した配列と磁気ディスクによる配列表に記録した配列が同一である旨の陳述書の提出があった。

4．　補正により、下記の書類が削除された。
☐ 明細書　第 　　ページ
☐ 請求の範囲　第 　　項
☐ 図面　図面の第 　　ページ／図

5．☐ この国際予備審査報告は、補充欄に示したように、補正が出願時における開示の範囲を越えてされたものと認められるので、その補正がされなかったものとして作成した。（ＰＣＴ規則70.2(c)　この補正を含む差し替え用紙は上記１．における判断の際に考慮しなければならず、本報告に添付する。）

国際予備審査報告

国際出願番号　ＰＣＴ／ＪＰ０３／１２８３９

Ⅴ．　新規性、進歩性又は産業上の利用可能性についての法第１２条（ＰＣＴ３５条(2)）に定める見解、それを裏付ける文献及び説明

１．　見解

新規性（N）	請求の範囲	2,3,7-20	有
	請求の範囲	1,4-6	無
進歩性（ＩＳ）	請求の範囲	15-20	有
	請求の範囲	1-14	無
産業上の利用可能性（ＩＡ）	請求の範囲	1-20	有
	請求の範囲		無

２．　文献及び説明（ＰＣＴ規則70.7）

文献１：若林一敏、外７名、“伝送用ＬＳＩを動作合成で開発，機能設計の期間が１／１０に短縮”、日経エレクトロニクス、日経ＢＰ社、1996.02.12、No.655、P.147-169

文献２：JP 2002-49652 A(安田博)2002.02.15、請求項１－５（ファミリーなし）

文献１に記載のＣｙｂｅｒは、動作記述言語ＢＤＬ（文法はＣ言語に似ている。）に基づいて高位合成を行いＲＴＬ記述を出力している。前記ＢＤＬはクロックの変化点（すなわち、クロック境界。）を「＄」により記述することができる。

ここで、＄と＄の間の変数への代入は、変数がどのレジスタにアロケートされるかは不明ではあるものの、レジスタへの代入を生じさせるから、レジスタ代入文と言える。

なお文献１における「＝」及び「＋＋」は「固有の演算子」と言える。

また、文献２には第２プログラム記述に相当するものが開示されている。

一方、ある言語で記述されたものを他の言語による記述にコンパイルすることは周知技術である。

してみれば、前記ＢＤＬを文献２に記載の記述にコンパイルすることは当業者が適宜なし得ることにすぎない。

Ⅷ.　国際出願に対する意見

請求の範囲、明細書及び図面の明瞭性又は請求の範囲の明細書による十分な裏付についての意見を次に示す。

請求の範囲１乃至２０項において、「レジスタ代入文」の技術的範囲が不明確である。

請 求 の 範 囲

1．（補正後）所定のプログラム言語を流用して記述された第1プログラム記述を回路記述に変換可能なコンパイラであって、

前記第1プログラム記述は、サイクル精度で回路動作を特定可能とする、固有の演算子が付されたレジスタ代入文とクロック境界記述を含み、

前記回路記述は、前記第1プログラム記述が特定する回路動作を実現するハードウェアを所定のハードウェア記述言語で特定することを特徴とするコンパイラ。

2．（補正後）所定のプログラム言語を流用して記述された第1プログラム記述を所定のプログラム言語を用いた第2プログラム記述に変換可能なコンパイラであって、

前記第1プログラム記述は、サイクル精度で回路動作を特定可能とする、固有の演算子が付されたレジスタ代入文とクロック境界記述を含み、

前記第2プログラム記述は、前のサイクルの状態を参照可能にする為に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文とを含むことを特徴とするコンパイラ。

3．（補正後）所定のプログラム言語を流用して記述された第1プログラム記述を、所定のプログラム言語を用いた第2プログラム記述と回路記述に変換可能なコンパイラであって、

前記第1プログラム記述は、サイクル精度で回路動作を特定可能とする、固有の演算子が付されたレジスタ代入文とクロック境界記述を含み、

前記第2プログラム記述は、前のサイクルの状態を参照可能にする為に前記レジスタ代入文を変形した変形代入文と、前記クロック境界記述

に対応して前記変形代入文の変数をサイクル変化に伴うレジスタの変化に対応させるレジスタ代入記述挿入文とを含み、

前記回路記述は、前記第２プログラム記述で定義されるハードウェアを所定のハードウェア記述言語で特定することを特徴とするコンパイラ。

４．前記所定のプログラム言語はＣ言語であることを特徴とする請求項１乃至３の何れか１項記載のコンパイラ。

５．前記ハードウェア記述言語はＲＴＬレベルの記述言語であることを特徴とする請求項１又は３記載のコンパイラ。

６．（補正後）タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とする、固有の演算子が付されたレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する第１処理と、

前記第１プログラム記述に基づいて前記タイミング仕様を満足する回路情報を生成する第２処理と、を含むことを特徴とする論理回路の設計方法。

７．前記第２処理は、前記第１プログラム記述を変換して、レジスタ代入文が入力変数と出力変数を用いて変形されると共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含む第２プログラム記述を前記回路情報として生成する処理を含むことを特徴とする請求項６記載の論理回路の設計方法。

８．前記第２処理は、前記第２プログラム記述を変換して、前記タイミング仕様を満足するハードウェアを所定のハードウェア記述言語で特定するための回路記述を更に別の前記回路情報として生成する処理を含むことを特徴とする請求項７記載の論理回路の設計方法。

９．前記プログラム言語はＣ言語であることを特徴とする請求項８記載

の論理回路の設計方法。

１０．前記第２プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含むことを特徴とする請求項９記載の論理回路の設計方法。

１１．前記第２処理は、前記第１プログラム記述を変換して、前記レジスタ代入文が入力変数と出力変数を用いて変形された記述を含む第２プログラム記述を前記回路情報として生成する処理を含むことを特徴とする請求項６記載の論理回路の設計方法。

１２．前記第２処理は、前記第２プログラム記述を変換して、前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含み、所定のプログラム言語で記述されてコンピュータで実行可能なむ第３プログラム記述を、前記回路情報として生成する処理を含むことを特徴とする請求項１１記載の論理回路の設計方法。

１３．前記第３プログラム記述を用いて設計対象回路のシミュレーションを行う第３処理を更に含むことを特徴とする請求項１２記載の論理回路の設計方法。

１４．（補正後）タイミング仕様に基づいて回路動作を定義するために、所定のプログラム言語を流用して記述され、サイクル精度で回路動作を特定可能とする、固有の演算子が付されたレジスタ代入文とクロック境界記述を含む第１プログラム記述を入力する入力処理と、

前記レジスタ代入文が入力変数と出力変数を用いて変形されると共に前記クロック境界記述に対応させて前記入力変数を出力変数に代入する記述を含み、前記所定のプログラム言語で記述された第２プログラム記述を生成する変換処理と、を含むことを特徴とする論理回路の設計方法。

１５．前記変換処理は、第１プログラム記述に基づいてＣＦＧを生成す